Online hardening of programs against SEUs and SETs

C. A. L. Lisbôa, L. Carro
Univ. Fed. do Rio Grande do Sul
Instituto de Informática
Porto Alegre, Brasil
{calisboa,carro}@inf.ufrgs.br

M. Sonza Reorda, M. Violante
Politecnico di Torino
Dipartimento di Automatica e Informatica
Torino, Italy
{matteo.sonzareorda,massimo.violante}@polito.it

Abstract

Processor cores embedded in systems-on-a-chip (SoCs) are often deployed in critical computations, and when affected by faults they may produce dramatic effects. When hardware hardening is not cost-effective, software implemented hardware fault tolerance (SIHFT) can be a solution to increase SoCs' dependability. However, SIHFT increases the time for running the hardened application, and the memory occupation. In this paper we propose a method that eliminates the memory overhead, using a new approach to instruction hardening and control flow checking during the execution of the application, without the need for introducing any change in its source code. The proposed method is also non-intrusive, since it does not require any modification in the main processor's architecture. The method is suitable for hardening SoCs against transient faults and also for detecting permanent faults.

1. Introduction

New semiconductor technologies, with shrinking feature sizes, allow the integration of more functions into a single chip than in the past, while higher clock frequency is increasing the number of operations that can be performed per time unit. Although all these benefits allow the implementation of highly efficient systems-on-a-chip (SoCs), those new technologies are very sensitive to soft errors. Therefore, when a SoC is intended for safety or mission-critical applications, designers must guarantee that soft errors have negligible impact on the behavior of the system.

SoCs are often designed resorting to intellectual property (IP) cores, which are usually only guaranteed to function correctly, while their correct behavior in presence of soft errors is normally not guaranteed. Therefore, it is up to the designers of safety or mission-critical SoCs to ensure that their systems are hardened against soft errors.

A possible approach to guarantee dependability of processor cores relies on the adoption of Software Implemented Hardware Fault Tolerance (SIHFT) techniques, based on modifying the software executed by the processor, introducing some sort of redundancy, so that errors are detected before they become failures. Such techniques are easy to use; however, their adoption is often limited by the high overhead they introduce in memory occupation and performance [1,2,3]. A hybrid approach with lower overhead, that combines software modifications with a special-purpose hardware module (known as infrastructure IP core, or I-IP core) is presented in [4].

SIHFT, and even this hybrid implementation, may increase memory occupation by a factor ranging from 2 up to 7. Redundant instructions are intrinsic to the concept of SIHFT and cannot be avoided; they can only be minimized, but at a cost of additional silicon area occupation [4]. In some types of applications, neither SIHFT nor its hybrid version are applicable, because of stringent costs or power budget. Moreover, when the SoC runs commercial off-the-shelf software components, SIHFT is simply not applicable, since the source code is not available. In this case, alternative techniques are needed for providing the system with an adequate level of dependability.

In this paper we propose a new technique to cope with, which combines ideas from SIHFT and hybrid techniques, and ensures *hardware and software transparency*. It exploits instruction hardening and consistency check, by performing twice each computation, and controlling the consistency of the attained results. The main novelties of this work are that the designer is freed from the need to modify the source code of the application, and that the implementation of the I-IP is non-intrusive, as far the main processor architecture is concerned, since the I-IP is located close to the main processor and performs instruction hardening, consistency checks, and control flow checking on-the-fly, by transparently modifying the sequence of instructions fetched by the processor. Each time a data processing instruction is recognized, the I-IP generates a sequence of instructions that are passed to the main processor, executes itself the data processing instruction and compares the results with what is computed by the main processor. It also recognizes the branch instructions the main processor fetches, and checks the correctness of the control flow execution.

Several advantages stem from this approach. Since the source code of the application running on the processor core is no longer needed, commercial off-the-shelf software components can be hardened, too. Because the application running on the main processor is hardened on-the-fly, no additional instructions need to be stored. Designers only have to guarantee the consistency of the information the SoC stores, for example by resorting to cost-effective codes like parity. The approach is scalable, allowing the designers to trade the area overhead and the performance degradation the I-IP introduces for the attained dependability level. Designers decide which of the instructions the processor executes have to be replicated on-the-fly by the I-IP, thus minimizing the amount of hardware resources needed to implement the I-IP. Finally, this approach can be used to detect the occurrence of transient faults (SEUs and SETs during normal operation), and also permanent faults (during manufacturing final tests).

The paper is organized as follows: Section 2 presents an overview of the available approaches to harden a SoC, Section 3 details the proposed approach, Section 4 presents an analysis of the approach, and Section 5 draws some conclusions.

2. Previous work

Error-detection techniques for software-based systems can be organized in three categories: software implemented, hardware based, and hybrid techniques. They focus on checking the consistency between the expected and the executed program flow, either by inserting additional code lines or by storing flow information in suitable hardware structures.

SIHFT techniques exploit the concepts of information, operation, and time redundancy to detect the occurrence of errors during program execution. Some of those techniques can be automatically applied to the source code of a program, thus simplifying the task of software developers: the software is indeed hardened by construction, and the development costs can be reduced significantly.

Techniques aiming at detecting the effects of faults that modify the expected program's execution flow are known as *control flow checking* techniques. These techniques are based on partitioning the program's code into basic blocks [5]. Among the most important solutions based on the notion of basic blocks proposed in the literature, there are the *Enhanced Control Flow Checking using Assertions* (ECCA) [7] and *Control Flow Checking by Software Signatures* (CFCSS) [1] techniques.

ECCA is able to detect all single inter-block control flow errors, but it is neither able to detect intra-block control flow errors, nor faults that cause an incorrect decision in a conditional branch. CFCSS cannot cover control flow errors if multiple nodes share multiple destination nodes. As far as faults affecting program data are considered, several techniques have been recently proposed that exploit information and operation redundancies [8,9]. The most recently introduced approaches modify the source code of the application to be hardened against faults by introducing information redundancy and instruction duplication, and adding consistency checks to the modified code to perform error detection. The approach proposed in [8] exploits several code transformation rules that require duplication of each variable and each operation among variables. The approach proposed in [9], named *Error Detection by Data Diversity and Duplicated Instructions* (ED^{4}I), consists in developing a modified version of the program, which is executed along with the original one. If results mismatch are found, an error is reported.

Both approaches introduce overheads in memory and execution time. The approach proposed in [8] minimizes the latency of faults; however, it is suitable for detecting transient faults only. Conversely, the approach proposed in [9] exploits diverse data and duplicated instructions, and thus is suitable for both transient and permanent faults. As a drawback, its fault latency is generally grater than in [8]. The ED^{4}I technique requires a careful analysis of the size of used variables, in order to avoid overflow situations.

Although very effective, SIHFT techniques may introduce time overheads that limit their adoption only to applications in which performance is not a critical issue. Also, in some cases they imply a memory overhead to store duplicated information and additional instructions. These approaches also require access to the source code of the application, precluding the use of commercial off-the-shelf software components.

Hardware-based techniques exploit special purpose hardware modules, called *watchdog processors* [10], to monitor the control flow of programs, as well as memory accesses. The behavior of the main processor running the application code is monitored using three types of operations. *Memory-access checks* monitor memory accesses executed by the main processor, such as in the approach proposed in [11], and activate an error signal whenever the main processor executes an unexpected access. *Consistency checks* of variables contents consists in controlling if the value stored in a variable is plausible. Watchdog processors can also validate each value the main processor writes or reads through range checks, or by exploiting known relationships among variables [12]. *Control flow checks* consist in controlling whether all taken branches are consistent with the Program Graph of the software running on the main processor [13,14,15,16].

As far as the control flow check is concerned, two types of watchdog processors may be envisioned. An *active watchdog processor* executes a program concurrently with the main processor. During program execution, the watchdog continuously checks whether its program evolves as that executed by the main processor or not [14]. This solution introduces minimal overhead in the program executed by the main processor; however, the area overhead needed for implementing the watchdog processor can be non-negligible. A *passive watchdog processor* does not execute any program; conversely, it computes a signature by observing the main processor's bus, and performs consistency checks each time the main program enters/leaves a basic block within the Program Graph. Passive watchdog processors are potentially simpler than active ones, but instructions needed for communicating with the watchdog introduce a performance overhead. Two alternative implementations are described in [15] and [16].

"*Dynamic verification*" (DIVA), another hardware based technique [6], uses a "functional checker" to verify the correctness of all computation executed by a core processor. It relies on a checker that is simpler than the core processor, because it receives the instruction to be executed together with the values of the input operands and of the result produced by the core processor, and the checker does not have to care about address calculations, jump predictions and other complexities that are handled by the core processor. Once the result of the operation is obtained by the checker, it is compared with the result produced by the core processor. If they are equal, the processor's result is forwarded to be written to the architected storage, otherwise the result calculated by the checker is forwarded. If a new instruction is not released for the checker after a given period, the processor executes again the instruction. Conceived as a solution to make fault-tolerant core processors, it evolved to use similar checkers to build self-tuning SoCs [18].

The DIVA approach cannot be implemented in SoCs based on FPGAs that have an embedded processor, because the checker is implemented inside the processor's pipeline. Also, it assumes that the checker never fails, due to the use of oversized transistors in its construction and extensive verification in the design phase. When this is not feasible, the use of alternatives such as TMR and concurrent execution with comparison, which have been already studied in several other works, is suggested.

Hybrid techniques such as [4] combine some SIHFT techniques with using an I-IP in the SoC. The software running on the processor core is modified to implement instruction duplication and information redundancy, and instructions are added to communicate with the I-IP. The I-IP works concurrently with the main processor, implements consistency checks among duplicated instructions, and verifies whether the correct program execution flow is executed. Such techniques are effective, since they provide a high level of dependability while minimizing the added overhead, both in terms of memory occupation and performance degradation, but they require the availability of the source code of the application.

The idea of introducing an I-IP between the processor and the instructions memory, and of making the I-IP replace on-the-fly the fetched code with hardened one, was preliminarily introduced in [19]. However, the I-IP proposed in [19] was much simpler, was not supported by a suitable design flow environment, as proposed here, its performance overhead was significant, and the method cannot cover permanent faults.

3. The proposed approach

Our approach minimizes the overhead needed to harden a processor core, with emphasis in the amount of memory used by the hardened application, and does not require access to the application's source code. It exploits the following concepts:

- *Instruction hardening* and *consistency check*: data processing instructions are executed twice, producing two results that are checked for consistency; and an error is notified whenever a mismatch occurs.
- *Control flow check*: each time the processor fetches a new instruction, the memory address is compared with the expected one, and an error occurs if they don't match.

The following subsections describe how these concepts are implemented.

3.1. Assumptions

The system we intend to protect is a SoC where a processor core is used to run a software application. We assume that a suitable I-IP core, able to implement instruction hardening and control flow checking, can be inserted in the SoC. The SoC can be either implemented through an ASIC, or an FPGA embedding a processor core.

The proposed I-IP is inserted between the memory storing the program code and the main processor core, and monitors each instruction fetch operation. In this work we assume that the instruction cache either does not exist, or is disabled. Moreover, we assume that the instruction memory and the data memory located outside the processor are hardened with suitable error detection/correction codes.

In developing our technique, we adopt the SEU in the processor's memory elements as the fault model. Possible fault locations are therefore the register file, and the registers not accessible through the instruction set (for example, the pipeline's boundary registers). However, the approach is general, and can be exploited to cope with other fault models, such as the single stuck-at or the single event transient ones.

3.2. Overall architecture

The technique we developed is based on inserting an I-IP between the processor core and the memory storing the instructions to be executed by the processor core, as Figure 1 illustrates. The I-IP implements the concepts of our technique as follows.

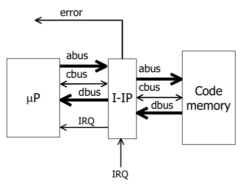

Figure 1. Overall architecture

Instruction hardening and *consistency check*: the I-IP decodes the instructions fetched by the processor. Each time a data processing instruction like that in Figure 2(a) is fetched, the I-IP sends to the processor the sequence of instructions in Figure 2(b). From the point of view of the processor, the fetched instructions are those issued by the I-IP. The sequence of instructions sent to the core processor includes two instructions that send the value of the source operands of the instruction.

```
FETCH_ADX: opcode dst, src1, src2
```
(a)

```
store I-IP-adx, src1
store I-IP-adx, src2
opcode dst, src1, src2
store I-IP-adx, dst
branch FETCH_ADX+OFFSET
```
(b)

Figure 2. (a) Original instruction (b) Source operands and result fetching

The third instruction (in boldface) is the instruction fetched from the program, while the fourth one is used to send to the I-IP the computed result. The last instruction is used to resume the original program execution, being `OFFSET` the size of the replicated instruction. Concurrently to the main processor, the I-IP executes the fetched instruction by using its own ALU, and compares the obtained result with that generated by the processor. In case a mismatch is found, the I-IP activates an error signal.

Control flow check: concurrently with instruction hardening and consistency check, the I-IP implements a mechanism to check if the instructions are executed according to the expected flow. Each time the I-IP recognizes a memory transfer, data processing, or I/O instruction stored at address A, it computes the address of the next instruction. Conversely, each time the I-IP recognizes the fetch of a branch instruction, it computes the address of the next instruction for the two cases corresponding to the branch taken (A_t) and to the branch not taken (A_n) situations. The former is computed taking into account the branch type, while the latter is computed as A plus the size of the branch instruction. When the next instruction is fetched, the I-IP controls if the program is proceeding along the expected control flow. If the fetch address differs from both A_n and A_t, an error signal is raised to indicate an unexpected address has been accessed.

Interrupt requests are not allowed to interfere with the execution of the sequence of instructions in Figure 2(b), since the I-IP receives interrupt requests through the IRQ signal, and forwards them to the processor core only after that sequence of instructions has been completely executed. This maintains correct operation of the I-IP, and allows to harden interrupt service routines at the cost of a slightly increased interrupt latency.

3.3. The I-IP

The I-IP we developed is organized as Figure 3 shows, and it is composed of the following modules:

1) *CPU interface*: connects the I-IP with the processor core. It decodes the bus cycles the processor core executes, and in case of fetch cycles it activates the rest of the I-IP.

2) *Memory interface*: connects the I-IP with the memory storing the application the processor executes. This module executes commands coming from the *Fetch logic* and handles the details of the communication with the memory.

3) *Fetch logic*: issues to the *Memory interface* the commands needed for loading a new instruction in the I-IP and feeding it to the *Decode logic*.

4) *Decode logic*: decodes the fetched instruction, whose address in memory is A, and sends the details about it to the *Control unit.* It classifies instructions in 3 categories:

a. *Data processing*: belongs to the set of instructions the I-IP is able to process. For those, the I-IP performs instruction hardening and consistency check. Moreover, for the purpose of the control-flow check, the address A_n of the next instruction in the program is computed, as described in 3.2.

b. *Branch*: may change the execution flow. The I-IP forwards it to the main processor and computes the two possible addresses for the next instruction, A_n and A_t.

c. *Other*: does not belong to the previous categories. The I-IP forwards it to the main processor and only computes the address A_n of the next instruction in the program.

5) *Control unit*: supervises the operation of the I-IP. Upon receiving a request for an instruction fetch from the *CPU interface*, it activates the *Fetch logic*. Then, depending on the information produced by the *Decode logic*, it either issues to the main processor the sequence of instructions summarized in Figure 2(b), or sends to the processor the fetched instruction. It also implements the operations needed for control flow check, and receives interrupt requests, forwarding them to the core at the correct time.

6) *ALU*: implements a subset of the main processor's instruction set. This module contains all the functional modules (adder, multiplier, etc.) needed to execute the data processing instructions the I-IP manages.

Two customization phases are needed for deploying successfully the I-IP in a SoC:

Processor adaptation: the I-IP is adapted to the main processor the SoC employs. This customization impacts on the *CPU interface*, the *Memory interface*, the *Fetch logic*, and the *Control unit*, and has to be performed only once, each time a new processor is adopted. Then, the obtained I-IP can be reused when the same processor is employed in a new SoC.

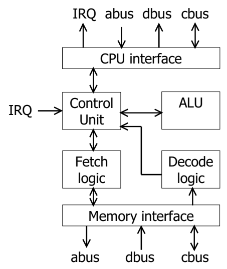

Figure 3. Architecture of the I-IP

Application adaptation: the I-IP must be adapted to the application executed by the main processor by adjusting the set of instructions to be hardened by the I-IP. This impacts the *Decode logic* and *ALU*. In this phase, designers can decide which of the instructions to be executed by the main processor, and how often, must be hardened. This phase may be performed several times during the development of a SoC, e.g., when new functionalities are added to the program running on the main processor, or when the designers tune the SoC area/performance/dependability trade-off.

3.4. The design flow

We developed the prototype of a tool that supports the automatic design flow depicted in Figure 4 to implement the application adaptation phase. A *disassembler* tool reads the binary code of the application that should run on the core processor of the SoC, and issues a report listing the instructions composing the binary code along with their frequency (i.e., how often an instruction appears) in the code.

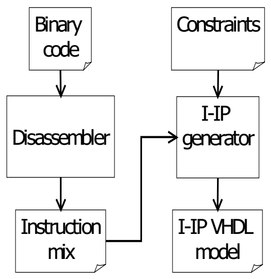

Figure 4. The design flow to support the application adaptation phase

The designer's constraints specify which instructions have to be hardened by the I-IP and which should be neglected (i.e., the I-IP forwards them directly to the main processor, without performing any other operation, except control flow check).

4. Experimental results

To assess the effectiveness of the proposed technique, we developed the prototype of a SoC composed of a processor core implementing the Intel 8051 instruction set, which runs a software implementation of the Viterbi algorithm for encoding a stream of data. We used Synopsys' Design Compiler to obtain a gate-level model of the SoC (using an in-house developed technology library). The SoC area occupation is 52,373 μm^2, including the processor core and its instruction/data memory.

In order to evaluate the impact of the adoption of the I-IP on the SoC design, we developed an Intel 8051 compatible version of the I-IP that implements the control flow check approach, as well as instruction hardening and consistency check. In order to evaluate the benefits stemming from the adoption of our technique, we evaluated its capability of detecting errors that may affect the SoC. In particular, we focused on SEUs affecting the processor core's internal memory elements. For this purpose, we performed several fault injection campaigns using the fault-injection environment presented in [17], and injected 10,000 randomly selected faults in the processor's memory elements (register file, control registers, and hidden registers embedded in the Intel 8051's control unit). During the experiments we used two different implementations of the I-IP's ALU, with different sets of instructions replicated. One replicates only increment instructions (the most used instruction in the program), while the other replicates both increment and add instructions. Table 1 lists the obtained results, in terms of percent reduction of failures (faults for which the outputs of the faulty SoC differ from the expected ones) observed with respect to the un-hardened SoC, area overhead the I-IP introduces, and performance overhead with respect to the un-hardened SoC. As can be seen, the approach increases significantly the dependability of the system. When increment instructions are hardened, the number of observed failures is reduced by 80%. When add instructions are hardened, too, the reduction of observed failures is about 88%. The figures concerning the area overhead and the performance degradation confirm that, the selection of instructions to harden allows designers to trade dependability for area increase and speed reduction. Please note that the added area is far below the amount of silicon area needed to duplicate the whole system, and also less than that for alternative approaches we adopted to harden the same system. When the approaches described in [8] are exploited, its area overhead is significantly higher than that of the one presented here, while the performance degradation and the fault detection capabilities of the two methods are comparable. When the approach in [4] is considered, its area overhead is higher than in the approach presented here. However, the approach in [4] has lower performance degradation and slightly higher fault detection capability. The method in [18] is not comparable, since it was implemented for a different processor. However, it is clearly characterized by a lower area overhead, a higher performance overhead, and a lower fault coverage.

Table 1. Attained results

Method	Instructions in the I-IP's ALU	Reduction of failures [%]	Area over-head [%]	Performance overhead [%]
Proposed Here	INC	81.3	13.1	292.0
	INC+ADD	87.5	15.7	314.0
Proposed in [8]	n. a.	81.8	76.2	388.3
Proposed in [4]	n. a.	92.5	51.8	108.9

Experiments also have shown that not all faults can be detected, since some failures have been observed for the hardened SoC. Some undetected faults affected memory elements that change the configuration of the processor core, e.g., they change the register bank select bit, switching from the used register bank, to the unused one. For this type of faults both the I-IP and the main processor fetch the operand for the instruction from a wrong source, and this type of fault escapes the error detection mechanisms provided by the I-IP. Some undetected faults affect the execution of branch instructions in such a way that the taken branch is consistent with the program control flow, but it is taken at the wrong time. A typical example is a SEU that occurs before a conditional branch is executed and affects the carry bit of the processor status word. In such a case, the wrong execution path is taken, based on a wrong value of the carry flag. However, the control flow is transferred to a legal basic block, which is consistent with the program's control flow, and therefore it escapes the control flow check that the I-IP employs. Finally, some undetected faults affected un-hardened instructions (mainly those concerning the return from a subroutine). As a final remark, it is worth noting that the method is able to detect all the permanent faults affecting the processor's *ALU*, since data manipulation instructions are executed in parallel by the I-IP.

5. Conclusions and future work

This paper presented a new approach to harden SoCs against transient errors in the embedded processor core, based on introducing in the SoC a further module, whose architecture is general, that needs to be customized to the adopted processor core. The I-IP monitors the processor bus and performs two main functions: when the processor fetches a data processing instruction belonging to a selected set, it acts on the bus and lets the processor fetch a sequence of instructions generated on-the-fly instead of the original one. The sequence of instructions allows the I-IP to get the operands of the original data processing instruction, which is then executed both by the processor and the I-IP; the results obtained by the processor and the I-IP are then compared for correctness. The I-IP also checks the correctness of the address used by the processor to fetch each new instruction, thus detecting a number of control flow errors. The method is inspired on SIHFT and hybrid techniques, but it does not introduce any memory overhead in the hardened system and does not require any change in the application code, whose source version is not necessary. Finally, the method allows designers to trade-off cost and reliability, by selecting the subset of instructions to be hardened.

Reported experimental results show that the approach increases the dependability of a SoC based on a processor core, while giving designers the possibility of trading off dependability with area overhead and performance degradation.

In the current implementation of the design flow, designers select the instructions to be hardened, according to their frequency in the code. We are now working to provide to designers profiling information coming from the analysis of the frequency of execution of each instruction during the evaluation of a representative workload.

6. References

[1] N. Oh, P.P. Shirvani and E.J. McCluskey. "Control flow Checking by Software Signatures", *IEEE Transactions on Reliability*, Vol. 51, No. 2, March 2002, pp. 111-112.

[2] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, M. Violante, "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.

[3] K. H. Huang, J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations", *IEEE Transactions on Computers*, vol. 33, December 1984, pp. 518-528.

[4] P. Bernardi, L. M. V. Bolzani, M. Rebaudengo, M. Sonza Reorda, F. L. Vargas, M. Violante. "A New Hybrid Fault Detection Technique for Systems-on-a-Chip", *IEEE Transactions on Computers*, Vol. 55, No. 2, February 2006, pp. 185-198.

[5] A. Aho, R. Sethi, J. Ullman, "Compilers: Principles, Techniques and Tools", Addison-Wesley, 1986.

[6] Austin, T. M. "DIVA: A Dynamic Approach to Microprocessor Verification". *The Journal of Instruction-Level Parallelism*, vol. 2, May 2000 (http://www.jilp.org/vol2).

[7] Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy, 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, No. 6, June 1999, pp. 627-641.

[8] 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 (part 3), December 2000, pp. 2231-2236.

[9] N. Oh, S. Mitra, E.J. McCluskey, "ED^{4}I: error detection by diverse data and duplicated instructions", *IEEE Transactions on Computers*, Vol. 51, No. 2 , Feb. 2002, pp. 180-199.

[10] A. Mahmood, E.J. McCluskey, "Concurrent error detection using watchdog processors-a survey", *IEEE Transactions on Computers*, Vol 37, No. 2, Feb. 1988, pp. 160-174.

[11] M. Namjaoo, E. J. McCluskey, "Watchdog processors and capability checking", in *Proceedings of the 12th International Symposium on Fault-Tolerant Computing (FTCS-12)*, 1982, pp. 245-248.

[12] A. Mahmood, D.J. Lu, E. J. McCluskey, "Concurrent fault detection using a watchdog processor and assertions", in *Proceedings of the IEEE International Test Conference 1983 (ITC '83)*, 1983, pp. 622-628.

[13] M.A. Schuette, J. P. Shen, "Processor control flow monitoring using signatured instruction streams", *IEEE Transactions on Computer*, Vol. 36, No. 3, March 1987, pp. 264-276.

[14] M. Namjoo, "CERBERUS-16: An architecture for a general purpose watchdog processor", in *Proceedings of the 13th International Symposium on Fault-Tolerant Computing (FTCS-13)*, 1983, pp. 216-219.

[15] K. Wilken, J.P. Shen, "Continuous signature monitoring: low-cost concurrent detection of processor control errors", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 9, No. 6, June 1990, pp. 629-641.

[16] J. Ohlsson, M. Rimen, "Implicit signature checking", in *Digest of Papers of the 25th International Symposium on Fault-Tolerant Computing (FTCS-25)*, 1995, pp. 218-227.

[17] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, M. Violante, "Exploiting Circuit Emulation for Fast Hardness Evaluation", *IEEE Transactions on Nuclear Science*, Vol. 48, No. 6, December 2001, pp. 2210-2216.

[18] Weaver, C; Gebara, F.; Austin, T. and Brown, R. "Remora: A Dynamic Self-Tuning Processor". *University of Michigan CSE Technical Report CSE-TR-460-02*, July 2002.

[19] M. Schillaci, M. Sonza Reorda, M. Violante, "A new approach to cope with single event upsets in processor-based systems", in *Proceedings of the 7th IEEE Latin-American Test Workshop – LATW 2006*, March 2006, pp. 145-150.