Analysis of the Effects of Soft Errors on Compression Algorithms Through Fault Injection Inside Program Variables

S. Avramenko, M. Sonza Reorda, M. Violante
Politecnico di Torino, Torino, Italy

G. Fey
German Aerospace Center (DLR), Bremen, Germany

Abstract—**Data logging applications, such as those deployed in satellite launchers to acquire telemetry data, may require compression algorithms to cope with large amounts of data as well as limited storage and communication capabilities. When commercial-off-the-shelf hardware components are used to implement such applications, radiation-induced soft errors may occur, especially during the last stages of the launcher cruise, potentially affecting the algorithm execution. The purpose of this work is to analyze two compression algorithms using fault injection to evaluate their robustness against soft errors. The main contribution of the work is the analysis of the compression algorithm susceptibility by attacking their data structures (also referred as program variables) rather than the memory elements of the computing platform in charge of the algorithm execution. This approach is agnostic of the downstream implementation details. Instead, the intrinsic robustness of compression algorithms can be evaluated quickly, and system-level decisions can be taken before the computing platform is finalized.**

Keywords—***Soft Errors; Compression Algorithm; Reliability; High-level Fault Injection;***

I. INTRODUCTION

Frequently, when considering space applications, the need arises for implementing compression and compaction algorithms able to reduce the size of the data to be sent to ground without compromising the amount of transferred information. The need for these algorithms is mainly driven by the limited amount of communication bandwidth existing from satellites or launchers to ground.

When implementing these algorithms, the designer may decide to adopt different solutions, ranging from purely hardware implementations (e.g., using ASICs or FPGAs), to purely software ones, using a suitably selected processing platform including a microprocessor, some memory, and some peripheral interfaces. Given the characteristics of space applications, in the selection of the most appropriate solution the reliability has to be particularly taken into account (alongside a number of parameters like power, performance, cost). The selection of the best solution is a complex process that is still highly based on the expertise of the designer and on the several constraints existing in the industrial environment he/she works in. However, the availability of proper tools able to analyze the characteristics of the different algorithms and to identify their advantages and disadvantages with respect to the different parameters, as well as to provide an estimate of their intrinsic robustness, may speed up the whole project flow. Based on this analysis, the designer is expected to select the most suitable algorithm, and then to decide about the platform that will take care of its implementation in space or on the launcher, receiving also mandatory indications about the most suitable fault tolerance mechanisms to be applied. Not surprisingly, it often turns out that the solution which optimizes one parameter (e.g., dependability) may be the one with the worst behavior in terms of another parameter (e.g., performance). Hence, the final choice is the one corresponding to the best trade-off among the different parameters, based on the project specifications.

Evaluating the dependability characteristics of an algorithm when no information about the target system is available yet, is a challenging task. Clearly the results of this evaluation must be used with care, serving mainly as a starting point for more detailed evaluations, that will be performed later in the design process. Any usage aiming at computing (or even estimating) statistically meaningful figures [5] about the dependability characteristics of the final product from this analysis must strictly be avoided.

From a technical point of view, such a kind of analysis typically adopts simulation-based fault injection [4], although formal techniques can also be applied in some cases [1][3].

Since in this early phase of the design process only the high-level code of each algorithm is available (e.g., in C or C++), the only feasible approach for performing this analysis is the one based on executing the corresponding program, injecting faults and observing the resulting behavior. In some cases, other representations of the algorithm may also exist (for example a Matlab's Simulink model): in these cases, a preliminary analysis can be performed by executing fault injection campaigns on this model [2].

When evaluating the dependability of an algorithm at high level and independently of the adopted hardware platform, a possible solution lies on adopting some virtualization platform (e.g., [10][11][12]). However, an even simpler solution (adopted in this work) corresponds to executing the algorithm code, and injecting faults in the program variables, finally checking the produced behavior and results. In this way, we can perform a first level analysis of the fault masking or self-convergence [13] capabilities of the code, as well as get some

978-1-5090-1331-9/16/$31.00 ©2016 IEEE

information about its intrinsic capabilities to detect possible errors.

An alternative solution to perform high-level dependability analysis is the one described and experimentally evaluated in [14], which is mainly based on the computation of the life-time of each variable. Although effective, that approach is completely static (i.e., the analysis is performed on the code of the application, without any form of execution), and is thus not able to take into account of the different usage and importance of variables.

This paper describes the work done for assessing the main characteristics in terms of dependability of several compression algorithms which are being considered for possible adoption on a launcher. The considered algorithms are intended to support the on-board processing and final transmission to ground of telemetry data gathered during the launcher flight. This work is performed without relying on any assumption on the target platform that will implement the algorithms, which is currently being defined. For performing dependability assessment we used two approaches.

The first approach we describe here uses the C version of the two algorithms, and performs fault injection campaigns where bit flips are applied to the variables of the programs. Finally, the method classifies the resulting fault effects. This raises some challenging issues about the selection of the variables and times for performing the injection, and provides results that can guide the designer in the following steps.

The second approach is based on an Instruction Set Simulator (in this case the target processor has been selected already) and fault injection is performed in its general purpose registers.

The goal of our work is to investigate whether the results gathered considering program variables can allow some ranking of the considered algorithms in terms of dependability. In particular, the obtained results provide indications about which algorithm is likely to produce the highest number of misbehaviors due to transient faults, and about the probability that faults are detected by intrinsic mechanisms, e.g., related to memory management. Clearly, the obtained figures can in no way forecast the final dependability figures of the final system, which also depend on the amount of memory used by each algorithm (which impacts on its sensitivity to radiation, which also depends on the possible mechanisms for protecting the different memory levels) and on its duration (the longer the time required to perform a given task, the higher the probability that a fault affects it [15]). However, they may allow to rank different algorithms according to their intrinsic robustness, and to draw suggestions for identifying and configuring the target hardware architecture.

The paper is organized in the following manner. An overview of the considered compression algorithms is given in Section II. Section III describes the Fault Injection environment we set up and discusses a few important choices that we faced when performing the selection of the faults to be injected. Section IV reports the gathered experimental results and outlines the indications that the designers can get from them. Section V draws some conclusions.

II. BACKGROUND ON COMPRESSION ALGORITHMS

Compression algorithms can be classified as:

- *Lossless* compression algorithms, that aim at minimizing the number of bits of the compressed output without any distortion of the original data, i.e., the decompressed bit stream is identical to the original bit stream. These algorithms are used in cases where accuracy of the information is essential.
- *Lossy* compression algorithms (also known as compaction algorithms), that aim at obtaining the best possible fidelity for a given bit rate or minimizing the bit rate to achieve a given fidelity measure. The decompressed bit stream is not identical to the original bit stream, although the resulting output is still meaningful for the considered applications, such as video or audio.

The techniques used to encode the original bit stream can be classified as *entropy encoding*, which leads to lossless compression, and *source encoding* that leads to lossy compression [6].

Entropy encoding refers to all those compression techniques that do not take into account the nature of the information to be compressed: entropy encoding techniques treat all data as a sequence of bits and ignore the semantics of the information to be compressed. On the contrary, source encoding techniques leverage the knowledge of the nature of the original bit stream. For example, if the original bit stream is audio or video, source encoding uses their inherent characteristics in achieving better compression ratio.

For the sake of this paper we considered only lossless compression algorithms. The application we are aiming at is indeed acquisition and compression of some telemetry data collected during the cruise of a satellite launcher. The data are crucial to analyze the behavior of the launcher, and can be fundamental to allow post-mortem analysis in case of launcher failure or provides insight into the launch that may be used for optimization. Therefore, some data must be acquired with maximum precision, and the original data must be reconstructed from the compressed bit stream without any distortion.

Among the possible entropy encoding techniques, in this paper we considered:

- *Dictionary-based techniques* [7] that take advantage of commonly occurring pattern sequences by using a dictionary in such a way that repeating patterns are replaced by a codeword that points to the index of the dictionary containing the pattern. Some of the data the telemetry system acquires can be handled effectively using dictionary-based techniques, because they are normally stationary to a bias value, and are subject to intense variations only rarely. As an example, we can consider data produced by the shock sensor: after lift-off where vibrations produced by the blast resulting from engine ignition are very relevant, shocks are stationary due to normal engine operations, and only during stage separation or orbit-injection of the payload significant signal changes are observed due to the blast producing stage/payload separation. For the sake of this paper we considered the

LZ78 (Lempel-Ziv 1978) dictionary-based technique, which achieves compression by replacing repeated occurrences of data with references to a dictionary that is built based on the input data stream, and in particular the LZW implementation [8].

- *Golomb-based techniques* [7] that exploit variable-length codes based on a model of the probability of the data dealt as natural numbers. To Golomb-code a number, we have to find the quotient and remainder of the division by a pre-defined divisor. The quotient is then expressed in unary notation, and the remainder is expressed in truncated binary notation. A Golomb-Rice code is a Golomb code where the divisor is a power of two, enabling an efficient implementation using shifts and masks rather than division and modulo. The interesting aspect of these techniques is that they are suitable for compressing streams of data, such as those acquired by the telemetry system, without using memory-demanding data structures such as dictionaries, and thus they are suitable when resource-constrained implementations are necessary. For the sake of this paper we considered the RICE techniques [9], which consist of a pre-processing phase that performs the subtractions between any two adjacent data words in the original bit stream. The actual encoding is performed on the result of this subtraction, using Golomb-Rice code.

III. THE EXPERIMENTAL SETUP

This section describes the experimental set-up we devised to analyze the impact of transient errors into compression algorithms.

The set-up is composed of a run-time environment and a fault injection environment, as described in the following sections, and depicted in Figure 1.

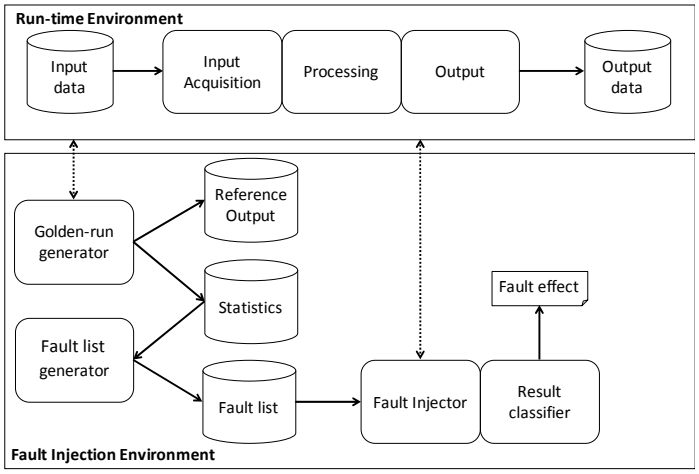

Figure 1. The experimental set-up

A. The run-time environment

The run-time environment is intended for executing a compression algorithm on a pre-defined set of input data. The software executed in the run-time environment is composed of the following modules:

- *Input acquisition module*: this module is in charge of loading from the file system the pre-defined set of input data in memory. This module makes use of the operating system API to read data from a binary file and to store them into a suitable input memory array.
- *Processing module*: this module is in charge of applying the selected compression algorithm on the input memory array, producing an output memory array. This module is coded in C language and does not make use of any operating system API. Thanks to this approach, the compression algorithm is highly portable, and can be reused in any target hardware platform (e.g., an embedded processor, as well as an ASIC/FPGA after high-level synthesis).
- *Output module*: this module is in charge of downloading the output memory array into the file system, using operating system API to create a binary file.

We implemented two versions of the run-time environment as described in the following:

- *Native run-time environment*, where the input acquisition, processing and output modules are executed through a Linux-powered x86 machine, which is also executing the fault injection environment.
- *Instruction set simulator (ISS)-based run-time environment,* where the input-acquisition, processing and output modules are executing through an instruction set simulator of an embedded processor. This implementation allows to validate the results obtained using the native environment and to capture hardware-specific behaviors. In this implementation the input acquisition and output modules are not used to avoid dependencies with any operating system. The processing thus makes use of an input memory array pre-initialized with the input data set, and runs on the simulated processor as bare metal code.

B. The fault injection environment

The fault injection environment we devised allows to evaluate the effects of SEUs, modeled as single bit flips, inside program variables in case of native run-time environment, or the simulated processor general purpose registers in the case of the ISS-based run-time environment. The fault injection environment is composed of four modules:

- *Golden-run generator*, which is in charge of running the compression algorithm once, using the run-time environment of choice to collect data to be later used for generating the list of faults to be injected as well as the reference output data set to classify fault effects. In particular, the golden-run generator collects:
  - Statistics about the data structure/registers the compression algorithm uses while processing the pre-defined set of input data. Only those data structures/registers that are read/written at least one time during data

compression are considered for fault injection. The rationale behind this approach is that algorithms may have data structures (such as large arrays) that are initialized once during execution (typically at the beginning), few entries of which are actually used to store meaningful data during algorithm execution. As a result, by avoiding injecting faults into unused data structures during algorithm execution we can save a significant amount of runtime.
    - The binary file storing the output memory array, which is used as reference for classifying the effects injected faults.
- *Fault list generator*, which is in charge of producing the list of faults to be injected. Based on the set of data structures of the compression algorithm (including both scalar variables such as temporary variables, indexes, as well as large data array) or the set of registers in the simulated processor, each defined as the tuple (*identifier name*, *size*), and the average execution time of the algorithm, the target generator produces randomly a predefined number of faults (i.e., the *fault list*), each defined as the tuple (*identifier name*, *bitmask*, *injection time*) where bitmasks correspond to the single bit in the selected identifier that must be bit-flipped at the injection time to model the effect of a SEU.
- *Fault injector,* which is a GDB-based script that for each fault starts the execution of the program in debug mode, advances the execution until injection time, injects the bit-flip in the identifier to attack, and resumes the execution of the program until its termination. The fault injection makes use of the run-time environment for compression algorithm execution. The selected ISS in case of ISS-based run-time environment shall thus support remote target connection to GDB. Fault injection takes place after the input acquisition module completed its activity, and before the processing module completed. As a result, faults are injected only during the execution of the compression algorithm.
- *Result classifier,* which is a shell script to analyze the termination status of the program under fault injection, and to compare the produced outputs with respect to a predefined reference (golden run).

Faults are classified according to the following categories:

- *Silent*: the faulty execution completes within a pre-defined amount of time and produces the same results as the golden run.
- *Timeout*: the faulty execution does not complete within a pre-defined amount of time.
- *Wrong output*: the faulty execution completes within a pre-defined amount of time and the produced results differ from the golden run.
- *Detected:* the faulty execution triggers some error detection mechanism. In the case of native run-time environment the exception is the segmentation fault produced as a result of a memory access violation. In the case of ISS-based run-time environment, the category is further detailed according to the exception the processor notified, that are:
    - *Bus error*, when the injected fault leads the program to access and invalid memory address.
    - *Alignment error*, when the fault leads the program to access a misaligned memory address.
    - *Illegal instruction*, when the fault leads the processor to access a memory area not containing valid instructions.

IV. Experimental Results

We performed our experiments on an Intel Core i7-powered workstation running a 64-bit Linux distribution. The run-time environments have been prepared to accommodate the execution of the two compression algorithms we considered: LZW and RICE. The characteristics of the two programs are reported in Table I, where the reader can find:

- The number of lines of code composing the two programs (*Lines of code*). We considered only the lines of code actually used to implement the compression algorithm, while we did not consider the lines of code for implementing the Input/Output operations needed to load into memory from the file system the data to be compressed, and to store the compression results to the file system.
- The *average execution time* for the two programs on the same set of input data.
- The *memory occupation* of the two programs. In this column we reported only the size of the data structure needed for the compression, while the input/output buffers have been neglected. From this column the much higher footprint of the LZW program is evident that, being based on a dictionary, requires a much higher amount of memory compared to RICE.

TABLE I. PROGRAM CHARACTERISTICS

	Lines of code [#]	Average execution time [µs]	Memory occupation [bytes]
RICE	180	2,753.00	64
LZW	110	1,827.00	21,091

As far as the ISS-based run-time environment is considered, we used the OpenRisc 1200 ISS as this soft-core is being considered as possible target hardware for our telemetry system.

We collected an initial set of results by injecting 100,000 randomly-generated faults into program variables (using the native run-time environment). The results we obtained are reported in Table II.

By analyzing the results, we can draw the following conclusions:

- As far as the faults classified as Silent are considered, we have that about 89% of faults fall in this category for LZW, while about 50% for RICE. This suggests the superior robustness of LZW versus RICE. However, the results achieved with this technique are highly inaccurate and could not be used to compute any quantitative comparison.
- As far as the faults classified as Timeout are considered, we can see that for LZW about 0.1% of the injected faults resulted in an abnormal program duration, while this is not the case for RICE. This result can be explained by considering the nature of the two compression techniques. Being based on a dictionary, LZW makes lookups in a large data array for each data word to be coded. As a consequence, in case the fault affects the index used for the search, it is possible that the number of iterations of the table lookup is affected greatly, thus leading to an abnormal program runtime. Conversely, RICE being based on simple arithmetic operations is insensible to such kind of effects.
- As far as the faults classified as Wrong Output are considered, we can see a substantial difference between the two programs, with faults producing about 11% of Wrong Output in LZW, and about 48% in RICE. These figures, although having no quantitative value, suggest that the cause of the eventual robustness superiority of LZW over RICE is the higher Wrong Output probability of RICE. Conversely, when considering the faults classified as Detected, we can see that about 0.02% of the injected faults fall in this category for LZW, while about 3% for RICE.

TABLE II. RESULTS FOR THE RANDOM FAULT INJECTION INTO PROGRAM VARIABLES

	LZW		RICE	
	[#]	[%]	[#]	[%]
Silent	88,982	88.98	49,330	49.33
Timeout	94	0.09	0	0.00
Wrong Output	10,906	10.91	47,834	47.83
Detected	18	0.02	2,836	2.84
TOTAL	100,000		100,000	

To characterize the considered algorithms and eventually validate the results of Table II, we performed a set of low-level injection experiments by exploiting the ISS-based run-time environment. The results we gathered by injecting 100,000 randomly selected faults in the OpenRISC 1200 general purpose registers (registers R1 to R31) are reported in Table III.

TABLE III. RESULTS RANDOM FAULT INJECTION INTO GENERAL PURPOSE REGISTERS

	LZW		RICE	
	[#]	[%]	[#]	[%]
Silent	79,661	79.66	51,410	51.41
Timeout	1,811	1.81	6,755	6.76
Wrong Output	6,078	6.08	24,866	24.87
Detected	12,450	12.45	16,969	16.97
TOTAL	100,000		100,000	

If we compare Table II and Table III, where in both cases randomly generated faults are considered with uniform probability, respectively for all data structures and all general purpose registers, we can see that:

- For all the parameters, with exception of Timeout, LZW is more robust with respect to RICE. The important aspect is the consistency of these figures between the two tables. In other words, the robustness figures obtained through the high-level analysis are consistent with the figures obtained through fault injection inside general purpose registers.
- The number of Timeout recorded during ISS-based fault injection does not respect the trend observed during native fault injection. This can be motivated by the fact that LZW has about 4 times the number of loops with respect to RICE. In particular, since the stack pointer R1 and link register R9 of the ISS-based run time environment can be affected by faults, LZW has a higher probability to have deadlocks.

To better investigate the effects of faults in the two algorithms at high level, a further approach of generating the target list has been used, in which the number of faults generated for a given identifier is proportional to the estimated number of times the identifier is referenced during the algorithm execution. The main goal of this approach was to avoid that the seldom used variables were considered with the same probability of heavily used ones. The results of these focused fault injection experiments are reported in Table IV.

TABLE IV. RESULTS FOR THE FOCUSED FAULT INJECTION INTO PROGRAM VARIABLES

	LZW		RICE	
	[#]	[%]	[#]	[%]
Silent	33,177	33.18	39,177	39.18
Timeout	1,466	1.47	0	0.00
Wrong Output	42,364	42.36	57,419	57.42
Detected	22,994	22.99	3,404	3.40
TOTAL	100,000		100,000	

As the reader can observe, there is limited consistency with the figures stemming from random fault injection. In particular, the evaluation about variables utilization has been considering the algorithm as it is, while the optimized compiled code makes the previous evaluation rather inaccurate. This aspect is

particularly visible for LZW, as it uses a huge number of variables.

Table V details the faults classified as Detected during general purpose registers random fault injection.

TABLE V. FAULTS CLASSIFIED AS DETECTED FOR RANDOM FAULT INJECTION INTO GENERAL PURPOSE REGISTERS

	LZW [%]	RICE [%]
Bus error	11.66	16.01
Alignment error	0.48	0.92
Illegal instruction	0.31	0.04
TOTAL (Detected)	12.45	16.97

Despite LZW shows twice the number of memory write accesses with respect to RICE (while the number of memory read accesses is the same), it is less affected by invalid/misaligned memory errors; this is likely to be due to its algorithmic peculiarities. Furthermore, LZW experiences a higher number of illegal instructions due to its higher usage of jump instructions.

V. CONCLUSIONS

Although quantitative evaluation approaches can provide better insight on prospective system architectures, system-level decisions are often taken resorting to the experience of designers due to the lack of adequate (i.e., fast, simple yet accurate) system-level analysis techniques. In this paper, we proposed a high-level analysis of two compression algorithms under investigation for a data logger system to be deployed in a satellite launcher.

Fault injection into the actually used program variables of two compression algorithms is used to evaluate the intrinsic robustness of the algorithms, as well as to draw some system-level considerations about the most suitable hardware architectures to implement each algorithm. Fault injection into the general purpose registers of a RISC architecture is used to validate the high-level considerations and to investigate hardware dependent characteristics of the algorithms.

The obtained results suggest that for two algorithms of the same class (e.g., compression algorithms), despite based on completely different paradigms, it is possible to perform a hardware independent robustness comparison which is consistent with the figures obtained through more precise techniques. It is well known that the fault tolerance estimation performed through high-level fault injection (e.g., injecting into program variables) is highly inaccurate [5]. This work means in no way to disprove this assumption, instead it suggests that under certain conditions it is possible to use high-level fault injection to rank algorithms based on their intrinsic robustness.

Our work is now exploring the possibility of improving the dependability characteristics of the algorithms by changing the code in order to harden it.

ACKNOWLEDGEMENTS

This work has been supported by the European Commission through the Horizon 2020 Project No. 637616 (MaMMoTH-UP).

REFERENCES

[1] V. C. Sharma, A. Haran, Z. Rakamaric, and G. Gopalakrishnan, “Towards formal approaches to system resilience,” in Proceedings of the 19th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC), 2013IEEE Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device, IEEE Std 1687-2014.

[2] H. Hakobyan, P. Rech, M. S. Reorda, M. Violante, “Early Reliability Evaluation of a Biomedical System”, in Proceedings of the 2014 9th International Design & Test Symposium, Algiers, Algeria, December 2014. pp. 45-50.

[3] M. Kooli and G. Di Natale, “A survey on simulation-based fault injection tools for complex systems,” in Proceedings of the 9th International Conference on Design & Technology of Integrated Systems in Nanoscale Era, DTIS 2014, Santorini, Greece, May 6-8, 2014, 2014, pp. 1–6.

[4] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, “Statistical fault injection: Quantified error and confidence,” in Proceedings of the Conference on Design, Automation and Test in Europe, ser. DATE ’09, 2009, pp. 502–506.

[5] Hyungmin Cho, S. Mirkhani, Chen-Yong Cher, J.A. Abraham, S. Mitra “Quantitative evaluation of soft error injection techniques for robust system design” in Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE

[6] D. J. C. MacKay, Information Theory, Inference and Learning Algorithms, Cambridge University Press, 2003

[7] D. Salomon, Data Compression: The Complete Reference, Springer-Verlag, 2007.

[8] T.A. Welch, “A Technique for High-Performance Data Compression, Computer”, vol. 17, no. 6, pp. 8–19, June 1984.

[9] “Lossless Data Compression, Recommendation for Space Data System Standards and D.C.C”. CCSDS 121.0-B-1. Blue Book, Issue 1. Washington, May 1997.

[10] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, “The Java Virtual Machine Specification”, Available at http://docs.oracle.com/javase/specs/jvms/se7/html/.

[11] Microsoft Corporation, “.Net Framework 4”, Available at http://msdn.microsoft.com/enus/library/vstudio/w0x726c2%28v=vs.100%29.aspx.

[12] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, ser. CGO ’04, 2004, pp. 75-86.

[13] R. Velazco, G. Foucard, F. Pancher, W. Mansour, G. Marques-Costa, D. Sohier, A. Bui, “Robustness with respect to SEUs of a self-converging algorithm”, in Proceedings of the 12th Latin American Test Workshop (LATW), 2011.

[14] A. Benso, S. D. Carlo, G. Di Natale, L. Tagliaferri, Prinetto, P., “Validation of a software dependability tool via fault injection experiments”, in Proceedings of the Seventh International On-Line Testing Workshop (IOLTW), 2001.

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