An Hybrid Architecture to Detect Transient Faults in Microprocessors

An Experimental Validation

Salvatore Campagna
DAUIN, Polytechnic of Turin
Torino, Italy
salvatore.campagna@polito.it

Massimo Violante
DAUIN, Polytechnic of Turin
Torino, Italy
massimo.violante@polito.it

Abstract—**Due to performance issues commercial off the shelf components are becoming more and more appealing in application fields where fault tolerant computing is mandatory. As a result, to cope with the intrinsic unreliability of such components against certain fault types like those induced by ionizing radiations, cost-effective fault tolerant architectures are needed. In this paper we present an in-depth experimental evaluation of a hybrid architecture to detect transient faults affecting microprocessors. The architecture leverages an hypervisor-based task-level redundancy scheme that operates in conjunction with a custom-developed hardware module. The experimental evaluation shows that our lightweight redundancy scheme is able to effectively cope with malicious faults as those affecting the pipeline of a RISC microprocessor.**

Keywords-tbd

I. Introduction

In recent years commercial off the shelf (COTS) components started appearing in applications domains where special-purpose parts were normally adopted. As an example, the traditional approach to design electronic equipment for space and avionic applications is based upon radiation-hardened devices (microprocessors, memories, ASICs, and FPGAs) that, by construction, are immune to the ionizing radiations that permeate the environment where the equipment has to operate [1]. As a matter of fact, although able to tolerate ionizing radiation, these devices are no longer able to satisfy the requirements of modern applications. New designs, like new aircrafts, as well as future space missions, impose to the designer requirements in terms of computing speed, and storage, which are possible to meet only through COTS components [2]. As a result, designers have to face a complex problem combining the need for high-performance which asks for COTS parts, and the need for radiation immunity which is typical of radiation-hardened parts.

As far as embedded computers for space and avionic applications are concerned, to solve the mentioned problem several radiation-hardened by design architectures and techniques have been proposed, based on the concepts of hardware redundancy, time redundancy, and information redundancy [3]. Triple-modular redundancy (TMR) is exploited in the SCS750 computer, where three COTS PowerPC processors are used in combination with COTS memory modules and radiation-hardened FPGAs implementing, among other functions, the majority voter and memory-coding algorithm [4]. Instruction-level time redundancy is used in the Proton series of single-board computers [5], where the multiple execution units available in COTS DSP or multi-core COTS processors are used to execute two times the same instruction and to vote among the obtained results for error detection. Instruction-level time redundancy is also exploited in other approaches targeting microcontrollers and single-core processors [6][7]. With these architectures lower power consumption and low hardware cost are possible with respect to TMR, although a time overhead has to be taken into account, and a special compiler is needed to enforce the time redundancy scheme the architecture exploits. Task-level redundancy is used in the DMT architecture proposed by CNES [8], where the same task is executed twice and a special-purpose memory bridge is used to avoid common-mode failures, as well as for error detection by checking the results produced by the replicated tasks. DMT does not require a specific compiler, however mandates that the software developer comply with a specially crafted design pattern, and it supports only a custom-made operating system. To give more flexibility to software designs, a novel approach was introduced recently that implements task-level redundancy through the use of virtual machines [9]. An hypervisor partitions the resources of an embedded computer and creates identical virtual machines running the same task. By running two instances of the same task and comparing the produced results (either through a dedicated lightweight virtual machine, or though custom hardware) error detection is achieved. Moreover, the hypervisor guarantees the segregation of each virtual machine in its own partition, hence preventing the faults spreading across replicated tasks. By using an embedded hypervisor, low overheads are achievable.

The purpose of this paper is to discuss an experimental investigation of the hypervisor-based task-level redundancy. In particular, we investigated on the behavior of the proposed architecture when the processor internal memory elements are affected by Single Event Upsets (SEUs) [10]. For this purpose we developed a novel fault injection method that allows emulating SEUs in the processor internal memory element (e.g., pipeline boundary registers) using an instruction set-simulator that does not include a detailed pipeline model.

The main outcomes of this paper are twofold. On the one hand, we experimentally validate the effectiveness of hypervisor-based task-level redundancy in dealing with malicious faults as SEUs in the processor internal memory elements; on the other hand, we present a novel fault injection system that allows approximating SEUs in the processor internal memory elements through an instruction-set model of the processor that does not include architectural details.

The rest of the paper is organized as follows. Section II defines the assumptions of our work. Section III recalls the hypervisor-based architecture introduced in [9]. Section IV presents the fault injection system we developed. Section V discusses the experimental results we performed, while Section VI draws some conclusions.

II. Assumptions

In space and avionic applications two types of embedded computers can be identified. The flight or platform computer is in charge of the critical functions of the system (i.e., keeping the proper orbit and orientation of the satellite, and managing the communication with the ground equipment), and the payload computer in charge of implementin
application functions of the system (i.e., collectir
a high-resolution camera, filtering and compres
delivering them to the platform computer for
These types of computers, depending on the mis
system is deployed, may have very different req
platform computer is normally performing a nu
but critical tasks (i.e., collecting inputs from low
executing a control algorithm, and drive suitabl
a result, it has low computing requirements
reliability and availability requirements. On th
payload computer executes perforn
computations that in general have lower
requirements: outages are allowed as they cor
loss of few data, while the system integrity is no
the high dependability/low performance
radiation-hardened processors and parts are
platform computers, while COTS devices are
new payload computers.

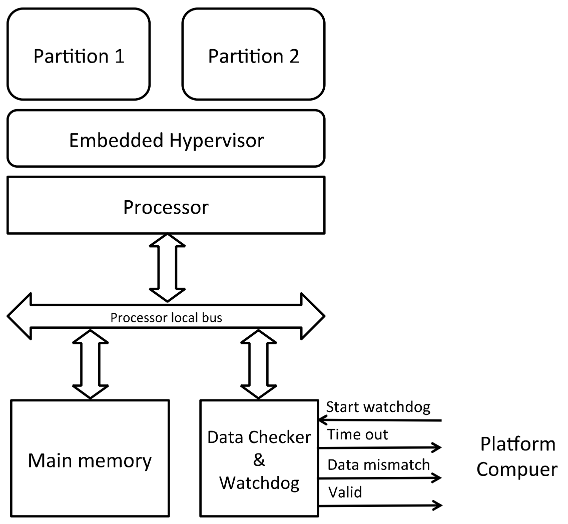
In this paper we discuss the experimental
solution to mitigate SEUs [10] that may affe
memory elements of COTS processors use
computers. In the system architecture, we assume the presence of two computers: a COTS-based payload computer, and a rad-hardened platform computer operating as data provider/data consumer for the payload computer, and as supervisor for the entire system (it initiates the operations of the payload computer and in case of error it implements the proper corrective actions). We assume that the COTS processor is based on RISC architecture, and that it embeds large memory blocks as register file and caches. Main memory is located in dedicated banks outside the processor chip, which are accessed though a proper memory interface the processor embeds. Consistently with modern COTS processor, we assume that large memory blocks, such as cache memories, are protected against SEUs using error detection and correction codes. Similarly, main memory is protected through error detection and correction codes. As a result, we assume that the targets of our investigation are SEUs affecting the remaining memory elements internal to the processor: register file, status/control registers, and hidden storage elements that are not accessible through the instruction set, i.e., the pipeline boundary registers.

In this work we assume that the payload computer executes an application organized in three phases:

- *Data acquisition*, during which the computer loads the input data from a buffer filled by the platform computer.
- *Data processing*, during which the computer applies suitable algorithms to the acquired data.
- *Data presentation*, during which the computer delivers the results to an output buffer read by the platform computer.

III. Hypervisor-based fault tolerance

The purpose of this section is to recall the concept of hypervisor-based fault tolerance for reliable embedded computing systems we introduced in [9]. The basic idea is to exploit task-level time redundancy by executing twice the task
e



As depicted in Figure 1, the COTS-based computer hardware runs an embedded hypervisor (EH), which is a tiny virtual machine monitor [11] in charge of partitioning and segregating the hardware resources. The EH executes two partitions according to a preemptive round-robin scheduler. Each partition is a virtual machine running a copy of the application task having access to a predefined amount of computing resources (the amount of time the partition can run on the processor, the amount of memory it can access, and the I/O ports it can use). While each isolated task executes, the EH identifies any access outside the allowed memory area and I/O ports. It thus provides mechanisms to avoid that a faulty partition corrupts the integrity of the whole system. As a result, the EH provides an execution environment for implementing seamlessly a time redundancy architecture, where two independent and segregated executions of the same task is achieved.

In order to provide error detection, a dedicated hardware module is exploited. It is a custom-developed Intellectual Property (IP) core that implements two functionalities: data checker and watchdog.

The IP operates as master for the payload computer local bus so that it can access the main memory independently from processor. Upon completion of the two tasks, the IP core compares the memory areas storing the results produced by the two partitions. In case of mismatch, an error signal is activated to inform the platform computer that a fault occurred during payload computation, so that the proper corrective action can be started; otherwise it generates a signal to inform the platform computer that the produced outputs are valid.

The IP implements a watchdog timer to avoid Single Event Functional Interruptions (SEFIs), which are situation where the payload computer becomes unresponsive, and does not produce any output to the platform computer. SEFI can be originated in two cases:

1. Data comparison is triggered when both the two partitions signaled to the IP their completion. In case one of the two never reaches the point where communications with the IP takes place, a SEFI can occur.
2. In case an SEU affects the processor during the execution of the embedded hypervisor, the system can be corrupted as the EH is not replicated. In this situation we expect that either the payload computer produces erroneous data (which are detected by the data comparison) or no data, resulting in a SEFI.

The platform computer starts the watchdog by issuing a start signal, triggering also the execution of the payload computer. In case of expiration, the IP issues a timeout signal to the platform computer that then initiates the proper corrective action.

IV. The developed fault injection approach

The purpose of the fault injection system is to investigate the behavior of a computing system in presence of a SEU, during the execution of a suitable workload. Several approaches are available in literature for fulfilling such a goal [12], ranging from accelerated testing to simulation-based fault injection.

In developing our fault injection approach we considered several aspects. We are looking for an approach that can be exploited by designers as a design aid, like a debugger or a HDL simulator. As soon as more and more COTS-based architectures are employed in critical applications, more and more often designer will have to rely on fault injection for debugging and validating their products. As a result, accelerated testing such as radiation testing and laser testing are not adequate.

The system under investigation is composed of a complex software stack encompassing one hypervisor and two partitions running on top of a modern COTS-based computer. The execution of the workload may thus take millions of clock cycles, while SEUs may hit in any of the memory elements the processor embeds. Although constraining our analysis to those memory elements that are not protected by any error detection and correction mechanism, the cardinality of the fault list can easily top tens of millions of faults. As a result, a very efficient fault injection method is needed for evaluating large fault lists in a moderate amount of time, which constraints the selection of the fault injection method to the following categories: software-implemented fault injection, emulation-based fault injection, and fast simulation-based fault injection as that possible through instruction-set simulators. Other approaches, like simulation-based fault injection though HDL simulators are not able to meet the required performance levels.

The system under investigation is based on a COTS processor that has been selected for performance reason. As a result, it is very likely that its architecture embeds advanced features (like pipeline, speculative execution, branch prediction, etc.) that have been introduced in the past decades to boost performance [13]. In order to inject SEUs in those memory elements used in such features either a detailed description of the processor is available (provided as synthesizable HDL that can be mapped on one or more FPGAs, or accurate architectural simulator), or a new approach is needed. As a matter or fact, unless for few exceptions, detailed descriptions of modern processors are not available, and therefore designers have to deal with instruction-set or system simulators [14][15] that offers behavioral models of the processor-based system, while neglecting most of the architectural details (e.g., pipeline structure).

In order to match the requirements of having a new design tool, fast execution speed, and capability of injecting SEUs in hidden memory elements, we developed a new fault injection system suitable for software-based fault injection (where an actual prototype of the system is available, and injection takes place in a actual processor chip), and simulation-based fault injection through instruction-set or system simulators.

For the sake of this paper we will limit our presentation to the technique we developed for inoculating SEUs in the processor pipeline using a processor model (either physical or simulated) that does not include a detailed pipeline model.

Analyzing the typical architecture of the processor pipeline we can find in any stage the following information stored in a suitable set of registers (pipeline boundary registers):

- R1: the op-code of the instruction in the pipeline stage, which is a group of bits defining the operation that the processor has to execute.
- R2: the source operands, which is a group of bits defining the data that has to be processed by the processor (this can be an immediate data, a register, or memory reference depending on the addressing mode).
- R3: the target operand, which is a group of bits defining where the result of the instruction has to be placed upon instruction completion (depending on the addressing mode and instruction, it can be a register or a memory reference).
- R4: the instruction result, which is a group of bits storing the outcome of the executed instruction that will be later

stored in the location specified as target operand, including the processor status word.

One SEU may affect one bit of these registers during the processing of a certain target instructions I_n in its evolution within the pipeline, starting from clock cycle T_f where I_n is fetched to T_{wb} where the result is committed.

In order to find a fault model to mimic the aforementioned SEU in a processor description that does not have the notion of pipeline, we considered the possible outcome of faults affecting the above-described registers.

A SEU affecting R1 produces visible effects if it changes the op-code right before the processor starts the instruction decoding phase, otherwise the SEU will be effect-less, as it will have no impact on the forthcoming operations based on a correctly decode op-code. This type of SEU can be approximated by bit-flips in the op-code of instruction I_n before it is fetched.

A SEU affecting R2 produces visible effects if it changes the source operands before the instruction reaches the pipeline stage where they are used (i.e., the stage where they are read from the register file, or fetched from memory, or where they are actually used in case of immediate addressing mode). Similarly, a SEU affecting R3 will have visible effects when the instruction I_n enters the pipeline stage where the result is committed to its final destination. These types of SEU can be approximated by bit-flips in those bits of I_n corresponding to source/target operands before it is fetched.

A SEU affecting R4 produces visible effects as soon as the result of instruction I_n is used by a forthcoming instruction I_{n+1}. In case I_{n+1} does not enter the pipeline before the committing of the result of I_n, the fault is propagated to the target destination for I_n. Otherwise, the faulty result is forwarded to I_{n+1} though the pipeline forwarding logic. The SEU in R4 is approximated as follows: the program trace is obtained (either though a debug pod in case of processor physical model, or using simulation commands), so that an ordered list of instructions is obtained according to the time instant when instructions enter the processor; then, the target instruction I_n is selected, and the instruction trace is parsed to find the instruction I_{n+1} using the result of I_n (which can be a general purpose register, the processor status word, any other special purpose registers, or a memory location); finally, a bit-flip is injected before the execution of I_{n+1} in the processor resource corresponding to the source operand of I_{n+1}.

One SEU affects instruction I_n as soon as it remains in the pipeline. If no effects are produced on the application, the next time I_n is fetched from the memory, the SEU will no longer be active. To model this transient behavior after the selection of the target where to inject, we operate as follows being I_n the target instruction:

1) The program is executed until I_n is about to be fetched.

2) The data and instruction caches are invalidated. This is necessary to force the processor to fetch the corrupted instruction or register from memory;

3) The fault selected according to the above model is injected by changing one bit of the code memory where I_n is stored, obtaining the faulty instruction I_n^* and the program execution resumed. In case of SEU propagated through the pipeline via the forwarding logic, the bit-flip is injected in the proper resource as explained before and $I_n^* = I_n$.

4) Right after the fetch of I_n^* the program execution is stopped, and the following operations are performed:

   a) The fault is removed from the code memory where I_n is stored, so that next time the SEU-free instruction will be fetched.

   b) The instruction cache is invalidated. This step is mandatory to avoid injecting multiple times the same SEU. Indeed, when entering the processor, I_n^* is stored in the instruction cache, and any following reference to the same instruction will lead to the execution of faulty copy of I_n.

5) The program execution is then resumed until program completion, or error/time-out detection.

Thanks to the proposed fault injection approach, it is possible to approximate the behavior of SEUs affecting the processor pipeline in a block-box testing fashion: a simple model of the processor is needed suitable for running an application program, being complex architectural-detailed models no longer needed. As a result, software-implemented fault injection can be used to inject SEUs in the processor pipeline, as well as simple behavioral instruction set simulators; moreover, by using less detailed but faster processor models, it is possible to greatly boost simulation-based fault injection.

V. EXPERIMENTAL RESULTS

The purpose of this section is twofold. On the one hand we present in section V.A a prototypical implementation of the hypervisor-based fault tolerant architecture, and of the proposed fault injection system. On the other hand, we discuss in Section V.B the experimental results we gathered during its evaluation.

A. Prototypical implementation

As proof of concept of our fault-tolerant architecture we developed a payload computer based on a modern processor that embeds advanced features: the LEON3 [16]. The processor has a 7-stage pipeline, separate data and instruction cache, and support up to four execution cores. In our architecture we employed a LEON3 with one core. As virtual machine manager, we adopted the Xtratum hypervisor [17], which is designed for supporting resource partitioning on the LEON processor family. Moreover, we coded in VHDL the data checker and watchdog timer required by the architecture depicted in Fig. 1. The checker described in [18] implements consistency checking on the produced results and SEFI detection with moderate hardware overhead (when synthesized on a Xilinx Spartan 3 FPGA, the checker requires 6% more LUTs than the LEON3 processor in our configuration, and 30% additional BRAM modules).

In developing an implementation of the fault injection system proposed in section IV we adopted TSIM as instructions-set simulator for the LEON3 system, and we developed a custom fault injection manager to orchestrate all the operations needed for running fault injection campaign: executing the program trace, parsing it to generate the fault list, run the injection of each fault in the fault list, and perform fault effect classification. For the sake of this paper we performed the injection of SEU in the LEON3 pipeline according to the methodology described in section IV.

Fault effects have been classified according to the following five categories:

- Silent: an SEU is classified as silent whenever at the end of the workload execution the valid signal is active, the data mismatch and time-out signals are not active, and the results produced by the two partitions are identical to those expected;
- Timeout: an SEU is classified as timeout when the timeout signal is active, i.e., whenever the workload execution is not completed after a predefined amount of time;
- Checker detected: an SEU is classified as detected by the checker whenever at the end of the workload execution the data mismatch signal is active;
- Hypervisor detected: an SEU is classified as hypervisor detected whenever it triggers any processor exception. In this case the exception can be internally managed by the hypervisor exception handling logic. This typically involves restarting the partition affected by the exception;
- Failure: an SEU is classified as failure whenever the results produced by the two partitions are equal but different from those expected, and neither the data mismatch nor the timeout signal is active.

B. Performed experiments

In our experiments we considered two applications running as hypervisor partitions:

- LZW: a universal lossless data compression algorithm applied over a buffer of 1024 bytes;
- JPEG-LS: another lossless data compression algorithm applied over a buffer of 1024 bytes;

Compression algorithms are very common for payload computers to process the data coming from on-board sensors. Compression algorithms are fundamental since the satellite becomes visible above the ground station for a limited amount of time during which the bandwidth usage must be optimized in order to be able to transfer as much data as possible. Moreover, on-board storage is limited, and therefore its usage must be optimized.

Table 1 summarizes the characteristics of the benchmarks in terms of code size, size of the output data, and number of executed instructions. In the table we report the total number of instructions needed to complete the workload, as well as the number of these instructions corresponding to hypervisor execution.

Table 1 – Benchmark characteristics

	LZW	JPEG-LS
Code size	147 Kbytes	170 Kbytes
Output data size	1,024 bytes	1,024 bytes
Num. of executed instructions	300,000	650,000
Num. of hypervisor instructions	30,085	3,115

As the reader can notice, the processor is busy running the hypervisor code for an amount of time spanning from 10% to less then 1% of the total workload execution time. These figures give an indication of the potential weakness of the fault tolerant architecture, as the hypervisor code is not replicated and any fault affecting the processor when running the hypervisor can lead to a SEFI, or can trigger hypervisor exceptions.

For each application a fault injection campaign has been executed. We collected the execution trace, and then for each instruction in the trace we generated one fault equivalent to SEUs in the processor pipeline as described in section IV. As the fault injection method requires invalidating the processor cache, a certain amount of time intrusiveness is expected. On the average (excluding the faults leading to timeout detection), we observed that the duration of the faulty run in less than 0.1% longer than the fault-free one. As a conclusion, for the considered application benchmarks, we claim that the time intrusiveness of the proposed fault injection method is negligible.

Table 2 reports the collected results in terms of fault classification.

Table 2 – Fault injection effects

	LZW	JPEG-LS
Injected faults	300,000	650,000
Silent	71.5%	72.2%
Timeout	1.3%	4.1%
Checker detected	2.8%	1.1%
Hypervisor detected	24.4%	22.6%
Failure	0.0%	0.0%

As shown in Table 2 the results for the two compression algorithms are consistent. Most of the faults are silent; while of those producing visible effects, the vast majority is detected by the hypervisor. The remaining are either classified as timeout or detected by the data consistency checker. In our experiment, although attacking randomly one instruction (belonging to either partitions or the hypervisor itself), we never recorded failures. The reason for most of the detected fault being caught by the hypervisor is that most of the instruction bits targeted by the injection specify either the instruction op-code or the addressing mode. As a result, the injected SEUs result in either invalid instructions, or memory accesses outside the boundary of the partitions.

Intuitively, by adopting a fault model mimicking the injection of SEUs in the processor pipeline, one would expect a lower number of silent faults than that recorded in our experiments. Although the results we achieved are consistent with other experiments published in [19] and performed on a similar architecture by an independent research group, we investigated the reasons for this result, and we found different scenarios:

- Some bits of the instruction are don't care, depending on the op-code and the addressing mode. As a result, by randomly selecting the bit to flip without knowing which are don't care bits, we introduce in the fault list effect-less faults. As an example we can consider case #1 and case #2 of Table 3. The instruction set of the LEON3 processor uses 32-bit instructions, where 9 bits specify the op-code. In case of registers are referenced in the instruction, each register is specified using 5 bits, and up to three registers can be used in one instruction. As a result, in case only one register is involved, 18 bits out of 32 are don't care, while in case of three registers are involved, 8 bits are don't care. This observation suggests that the fault list generation process should be modified to avoid injecting into don't care bits.
- In some cases the faulty instruction, although being different from the fault-free one, behaves in the same way. As an example we can consider case #3 in Table 3, where the `clr` and `st` instructions produce the same results.
- In some cases the architectural state of the processor is such that, although performing the same operation among different input data, the faulty and the fault-free instructions produce the same result, as in case #4 of Table 3.

Table 3 - Silent effects

	Original Instruction (binary code) instruction	Flipped Instruction (binary code) instruction
#1	`(80a20009) cmp %o0, %o1`	`(80a20089) cmp %o0, %o1`
#2	`(01000000) nop`	`(01000400) nop`
#3	`(c0220000) clr [%o0]`	`(c0220100) st [%l0], [%o0]`
#4	`(90022004) add %o4, 4, %o4`	`(90022014) add %l4, 4, %o4`

VI. CONCLUSIONS

In the near future, more and more applications will benefit from the adoption of COTS components, and processors in particular, in fields where fault tolerance is a dominant requirement. Fault tolerant architectures as well as fault validation tools for COTS-based designs will thus become mandatory.

In this paper we presented a novel fault injection environment that can be used by designers in their development flow with minimal set-up effort, as it is based on either an instruction set simulator (which is nowadays a commonly available feature provided by processor manufacturers), or the actual processor chip. The attained results injecting faults simulating SEUs in the processor pipeline are consistent with previously published results by an independent resource group, thus suggesting the soundness and accuracy of the approach.

Through the proposed environment we performed the injection of faults that model SEUs affecting the pipeline of a processor running a complex software-based architecture for error detection, which exploits hypervisor-based task-level redundancy. The results show empirically the robustness of the architecture.

REFERENCES

[1] Barth, J.L.; Dyer, C.S.; Stassinopoulos, E.G.; "Space, atmospheric, and terrestrial radiation environments", Nuclear Science, IEEE Transactions on , vol.50, no.3, pp. 466- 482, June 2003

[2] Pignol, M.; Malou, F.; Aicardi, C.; "Qualification and relifing testing for space applications applied to the agilent G-Link components", On-Line Testing Symposium (IOLTS), 2010 IEEE 16th International, pp.103-108, 5-7 July 2010

[3] Koren, I.; Krishna, C. M.; Fault-Tolerant Systems, Morgan Kaufmann, 2007

[4] www.maxwell.com

[5] www.spacemicro.com

[6] Bernardi, P.; Bolzani, L.M.V.; Rebaudengo, M.; Reorda, M.S.; Vargas, F.L.; Violante, M.; "A new hybrid fault detection technique for systems-on-a-chip", Computers, IEEE Transactions on , vol.55, no.2, pp. 185-198, February 2006

[7] Azambuja, Jose Rodrigo; Lapolli, Angelo; Altieri, Mauricio; Kastensmidt, Fernanda Lima; "Evaluating the efficiency of data-flow software-based techniques to detect SEEs in microprocessors", Test Workshop (LATW), 2011 12th Latin American, pp.1-6, 27-30 March 2011

[8] Pignol, M.; "DMT and DT2: two fault-tolerant architectures developed by CNES for COTS-based spacecraft supercomputers", On-Line Testing Symposium, 2006. IOLTS 2006. 12th IEEE International, July 2006

[9] Campagna S., Hussain M., Violante M.; "Hypervisor-Based Virtual Hardware for Fault Tolerance in COTS Processors Targeting Space Applications", Defect and Fault Tolerance in VLSI Systems, pp. 44-51, October 2010.

[10] Dodd, P.E.; Massengill, L.W.; "Basic mechanisms and modeling of single-event upset in digital microelectronics", Nuclear Science, IEEE Transactions on , vol.50, no.3, pp. 583- 602, June 2003

[11] Goldberg, R.; "Survey of virtual machine research", IEEE Computer Magazine, vol. 7, no. 6, pp. 34–45, 1974

[12] Benso, A.; Prinetto, P.; "Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation", Springer, 2003

[13] Hennessy, J. L.; Patterson, D. A.; "Compuetr Architecture: A Quantitative Approach, 4th editorion", Morgan Kaufamann, 2006

[14] TSIM LEON/ERC32 System Simulator, Gaisler Aeroflex

[15] http://www.windriver.com/products/simics/

[16] www.gaisler.com

[17] Crespo, A.; Ripoll, I.; Masmano, M.; "Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach", Dependable Computing Conference (EDCC), 2010 European, pp.67-72, 28-30 April 2010

[18] Campagna S., Hussain M., Violante M.; "A Light-Weight Fault Tolerance Framework for Space Computing using COTS Components", Lecture Notes in Computer Science (2011)

[19] Touloupis, E.; Flint, J.A.; Chouliaras, V.A.; Ward, D.D.; "Study of the Effects of SEU-Induced Faults on a Pipeline Protected Microprocessor", Computers, IEEE Transactions on , vol.56, no.12, pp.1585-1596, December 2007