A New Mitigation Approach for Soft Errors in Embedded Processors

Francesco Abate, Luca Sterpone, and Massimo Violante

***Abstract*—Embedded processors, like for example processor macros inside modern FPGAs, are becoming widely used in many applications. As soon as these devices are deployed in radioactive environments, designers need hardening solutions to mitigate radiation-induced errors. When low-cost applications have to be developed, the traditional hardware redundancy-based approaches exploiting m-way replication and voting are no longer viable as too expensive, and new mitigation techniques have to be developed. In this paper we present a new approach, based on processor duplication, checkpoint and rollback, to detect and correct soft errors affecting the memory elements of embedded processors. Preliminary fault injection results performed on a PowerPC-based system confirmed the efficiency of the approach.**

***Index Terms*—Embedded processors reliability, fault injection, single event effects.**

I. Introduction

DEPENDABLE processors able to tolerate Single Event Effects (SEEs) are needed when designing safety- or mission-critical applications. This problem is subject to intensive researches, which follow two main research paths. Earlier solutions consisted in developing radiation-hardened processors [1] by using special manufacturing technologies, as well as suitable fault-immune architectures. Although effective, these solutions are very expensive since they rely on special-purpose versions of the processors, and therefore they can be used only in those applications where cost is not a primary concern (e.g., military applications). More recent solutions consist in the adoption of particular fault-tolerance design techniques to develop the systems where the processors are deployed that guarantee a given degree of dependability at a cost of low area and performance overheads, while not introducing any modification to the processors themselves. Therefore, although introducing some penalties, these approaches are far more cost-effective than the former ones, since low-cost general-purpose processors can be employed.

Several fault-tolerance design techniques have been developed in the recent years in order to increase the dependability of processors, with particularly emphasis to the issues of fault detection. These techniques can be classified as software-based and hardware-based ones.

Software-based approaches detect faults within the program's control flow or in the program's data, and also provide coverage of those faults that affect the memory elements the processor embeds (e.g., the processor's state register, or temporary registers used by the arithmetic and logic units) [2], [3]. The main benefit stemming from software-based approaches is that fault detection is obtained by modifying the software running on the processor, only, by introducing instruction and information redundancies, and consistency checks among replication computations. The increased dependability comes at a cost of memory (for the additional data and instructions) and performance overheads (for the replicated computations, and for the consistency checks) that may not be acceptable in certain type of applications.

Hardware-based techniques consist in introducing in the system redundant hardware to make the whole more robust against SEEs. Some researches proposed to attach special-purpose hardware modules known as watchdogs to the processor in order to monitor the control-flow execution, the data accesses patterns [4], and to perform consistency checks [5], while letting the software running on the processor mostly untouched. Although watchdogs have limited impact on the performance of the hardened system, they may require non-negligible development efforts. Moreover, watchdogs are barely portable among different processors. To combine the benefits of software-based approaches with those of hardware-based ones, a hybrid fault detection solution was introduced in [6]. This technique combines the adoption of software techniques in a minimal version, for implementing instruction and data redundancy, with the introduction of an Infrastructure-Intellectual Property (I-IP) attached to the processor, for running consistency checks. The I-IP behavior does not depend on the application the processor executes, and therefore it is widely portable among different applications.

Other researches explored alternative paths to hardware redundancy, which consisted basically in duplicating the system's processor and inserting special monitor modules that check whether the duplicated processors execute the same operations [7], [8]. These approaches are particularly appealing in those cases where processor duplication does not impact severely on the hardware costs. Moreover, since they do not require modifications to the software running on the duplicated processors, as conversely happens for the software-implemented approaches, commercial off the shelf software components can be hardened seamlessly.

The recent years have seen the introduction of processors inside FPGA devices, and few products are now available (e.g., Xilinx Virtex-II Pro FPGAs, Virtex-IV and 5) that allow implementing on a single device complex hardware/software systems where the software runs on the embedded processor, which communicates with custom hardware implemented by the FPGA's

Manuscript received September 7, 2007; revised December 12, 2007. Current version published September 19, 2008. This work was supported in part by the Italian Ministry for University and Research (MIUR) and Regione Piemonte through the FELIx project.

The authors are with Politecnico di Torino, 10129 Torino, Italy (e-mail: massimo.violante@polito.it).

Digital Object Identifier 10.1109/TNS.2008.2000839

logic resources. Due to their characteristics, hardware based approaches, as that introduced in [7], [8], are well suited for hardening the processor cores FPGAs embed. Devices can be found including two processors. Moreover, glue logic can be implemented easily using the FPGA fabric.

The main contribution of this paper is the introduction of a new approach based on rollback recovery using checkpoints for hardening processor cores inside FPGAs. The approach takes its roots from [8], and introduces the possibility of surviving the occurrence of soft errors. In our approach a checkpoint feature is introduced to save the state of the processor, along with a rollback feature that makes possible resuming executing the software from a previously known safe state (saved during checkpoint) each time an error is detected.

We implemented the proposed technique targeting the PowerPC 405 processor embedded in Virtex II pro devices, and we validated it through extensive fault injection results that showed the effectiveness of the proposed approach.

The rest of the paper is organized as follows. Section II introduces the concepts of checkpoint and rollback, then it describes how we exploited such concepts in our approach. Section III presents an experimental evaluation of our approach on a prototypical systems. Finally, Section IV draws some conclusions.

II. Rollback Recovery Using Checkpoints

This section described the basic concepts of rollback recovery using checkpoints, and then it presents the approach we developed to mitigate SEE affecting processors embedded in FPGAs.

A. Basic Concepts

Rollback recovery using checkpoints is known to be a cost-effective approach to implement fault tolerance in processor-based systems [9].

The approach is based on the following concepts:

- *Context*, which can be defined as the set of information needed to univocally define the state of a processor-based system (it can include the content of the processor's registers, the cache, the main memory, etc.).
- *Checkpoint*, which can be defined as the operation needed to retrieve and store the system's context.
- *Rollback*, which can be defined as the operation that sets the state of a systems to a context saved during checkpoint.

Checkpoint and rollback can be exploited to survive the occurrence of soft errors as depicted in Fig. 1.

After the program executed for a predefined amount of time, for example a fixed amount of time, or until a milestone during program executing is reached (e.g., a value is ready for being committed to the program user, or a value is ready for being written in memory) a sanity check is executed.

In case the sanity check is passed, indicating that the processor-based system is fault-free, a checkpoint is executed, thus storing the context in a safe storage (i.e., a set of memory modules which cannot be altered by soft errors).

In case the sanity check fails, indicating the detection of a soft error, rollback is performed, thus leading the system to a previously known good state. In this case, the computation the program executed before the sanity check fails is repeated.

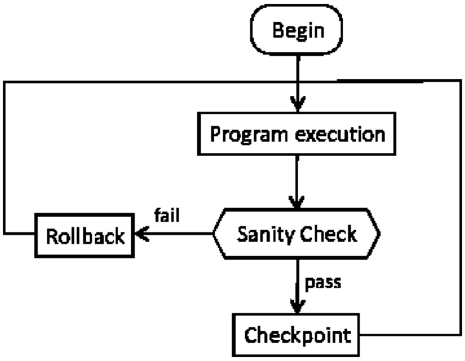

Fig. 1. Flow chart of rollback recovery using checkpoint.

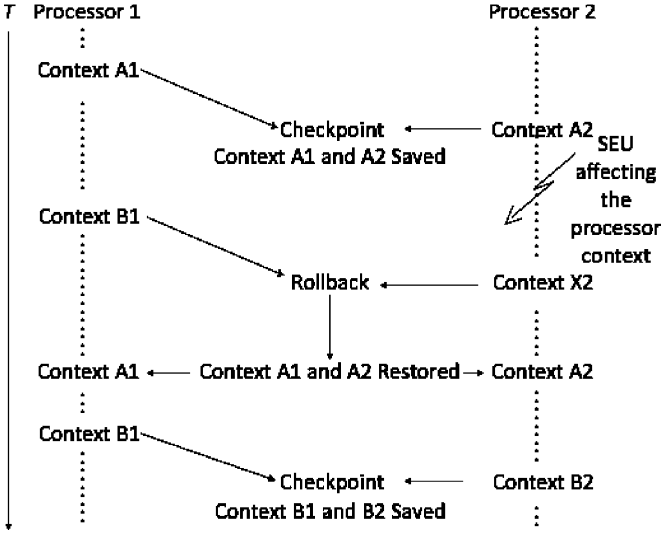

Fig. 2. Example of execution of rollback recovery using checkpoint.

An example of the execution flow of rollback recovery with checkpoint is shown in Fig. 2. According to the timeline indicated by the arrow on the left, the Processor1 first and the Processor2 later, execute the same portion of the application code in handshake, reaching the processor contexts A1 and A2 in two different times. As the two contexts are the same no error is detected and both contexts can be saved in a safe storage. The execution can then restart and the Processor 1 executes a new application code section reaching the context B1, it stops its execution, and the Processor 2 can start the execution of the same application code section. As a SEU occurs during its execution, it reaches the context X2 different from B1 of Processor 1. The difference in the two contexts indicates an error and, as consequence, a rollback has to be performed. Such operation restores the two processors context to the last one saved as safe, i.e., the contexts A1 and A2. Executing again the same application code in handshake fashion, the Processor 1 and Processor 2 reach the context B1 and B2, the right ones. Thus the SEU fault has been detected and tolerated.

The implementation of the approach is not trivial, as several issues must be addressed.

A criteria to interrupt the processor and execute the sanity check has to be identified. This choice is particularly critical as it can have severe impact on the system's performance, as well as on the fault latency. On the one hand, in order to minimize fault latency, sanity checks should be performed as frequently as possible. On the other hand, interrupting frequently the program execution may have unacceptable drawbacks on the system's performance. Therefore, a suitable trade-off must be established, which should take into account the cost for implementing the sanity check as well.

A suitable sanity check has to be defined. The goal of the sanity check is to identify whether faults are present in the system. Therefore, it plays a very important role in the analyzed fault-tolerance approach. Maximum fault identification capability should be provided, while minimizing the sanity check implementation costs.

The processor's context has to be defined. The designers should identify the minimum set of data that allows the system to be restored to a correct state in case of fault detection. As the amount of data in the context affect the cost for implementing checkpoint and rollback, trade-off must be used to balance the implementation costs with the overhead required at each checkpoint/rollback.

A safe (fault-immune) storage for the context has to be defined. As the processor's context saved during checkpoint will be used in case of fault to recover the correct state of the system, the memory storing it should be bulletproof against the fault model of concern.

The procedures to efficiently implement the checkpoint and rollback operations have to be developed. Both operations require to access to all the memory elements defining the processor's context, and must be performed every time the context has to be retrieved and stored (checkpoint) or loaded in the processor (rollback). Depending on the definition of the context, the frequency of sanity check execution, as well as the fault occurrence frequency, checkpoint/rollback operations may be performed very frequently, and therefore the time spent while moving data to and from the processor must be minimized.

B. Synchronized Lockstep With Rollback

The synchronized lockstep with rollback is conceived to harden processor macros inside FPGA devices against soft errors affecting the memory elements they embed, and it is inspired to the approaches presented in [7] and [8].

In its current implementation, the approach is tailored to the PowerPC processor the Virtex II pro, Virtex-IV and Virtex 5 devices embed. However, the approach is general and it can be extended to different embedded processors (e.g., the ARM one embedded in Actel devices). Moreover it can be adapted to be integrated with [8].

1) Sanity Check Implementation: Due to the availability of two processor cores in the considered devices, we adopted processor duplication with output comparison to implement the sanity check. The approach we developed exploits two processors running the same software. As the processors are synchronized, and executing the very same software, they are expected to perform the very same operations. By observing the information travelling to and from the processor (i.e., by monitoring the two processors' data, address, and control buses) it is therefore possible to identify fault-induced misbehaviors. In our implementation the sanity check consists in comparing the content of the data, address and control buses of the two processors.

The sanity check is performed every time the two processors perform a write cycle, thus every time they send information either to the memory or the peripheral devices. When no mismatches are found between the two processors' buses, we assume that sanity check to be passed successfully, and we proceed storing the checkpoint.

The main difference between our approach and alternative ones targeting the same devices (e.g., [8]) is to let the processors run alternately in a hand shake fashion: one processor executes the software until a write instruction occurs; it then stops the execution, and waits for the second processor to execute the very same software. As soon as the second processor executed a write operation, it is stopped, and the first one resumes executing the software. We refer to the time spanning between the start of the execution of the first processor, and the completion of the write operation by the second processor as *execution cycle*.

Being the two processors executing the same software, at the end of one execution cycle they are synchronized, i.e., both of them have performed the very same write operation, writing the same value to the same memory address. Any difference in the written value or in the address, or in the control lines, signals the presence on an error. Therefore, in our approach at the end of each execution cycle, a dedicated hardware module checks for the consistency of the write operations issued by the two processors. In case no mismatch is found, we assume that both processors are not affected by faults, and we save the status of the memory elements of both processors in a dedicated memory, thus implementing the checkpoint operation storing the system's context.

2) Context Definition: For the sake of this paper the context contains all the information needed to define the status of the processor (content of user and special registers, program counter, stack point, processor' status word, etc.). As a limitation of the current implementation of our approach, we are not able to save the status of the processor's cache, and therefore we assume it is disabled. Moreover, we do not include the processor's main memory in the context. As a side effect of this choice, the application should be written in such a way that input data are not modified during the execution cycle, so that in case of rollback the integrity of input data can be preserved.

3) Context Storage: For the sake of this paper we assume the memory used to store the processor's context immune to SEUs. We thus assume that it is hardened using suitable EDAC codes as well as memory scrubbing.

4) Overall Architecture: The architecture of the proposed system includes the following modules as depicted in Fig. 3:

- *PPC0* and *PPC1*: the two processors working in hand shake.
- *Interrupt IP0* and *Interrupt IP1*: as detailed later, they are in charge of issuing an interrupt request to trigger the saving or the restoring of the processors' context.
- *Opt_intc0* and *Opt_intc1*: the interrupt controllers attached to PPC0 and PPC1, needed to manage the interrupt requests coming from Interrupt IP0 and Interrupt IP1.

- *Program code controller IP*: the module in charge of managing the synchronization of the two processors, and performing error detection.

Processor synchronization is implemented using the PPC0/1's *halt* signal to pause the software execution. Error detection is implemented by interfacing with the PLB memory controller though which the PPC0/1 issues read and write requests to the data memory. When comparing two write operations issued by PPC0 and PPC1, in case no mismatch is found the savecontext signals is issued to both Interrupt IPs; otherwise, the rollback signal is issued. The savecontext signal means that the Interrupt IP module has to save the process's context, including both general and special purpose register, while the rollback one forces the Interrupt IP to execute the interrupt routine that restores the previously saved processor's context, i.e., the last one considered as safe.

5) Checkpoint and Rollback Implementation: Checkpoint and rollback operations are particularly critical, as they may have severe impact on the system's availability. As the system is off-line when checkpoint and rollback are executed, any new input cannot be readily consumed and elaborated. As a result, deadline can be missed, or in case input buffers are not large enough some data may be lost.

In the following we describe how we implemented checkpoint and rollback in our approach. It is worth to remark here that, as in our case the context is defined by the memory elements of the processor only, we do not deal with the cache or external memory.

In order to minimize the time needed for checkpoint and rollback execution, we implemented them using the interrupt mechanisms the considered processor set available.

Usually, when an interrupt request is processed the following operations are performed:

- The processor saves its context into the stack and starts executing the interrupt routine.
- When the interrupt routine ends, the processor restores its context from the stack and continues the execution.

Our implementation of checkpoint takes advantage of this mechanism as follows:

- Upon receiving the savecontext signal, an interrupt request is issued to the processor.
- When executed, the interrupt routine copies the content of the stack storing the processor's context in a dedicated memory, the *context memory*, and quits.

Rollback operation consists in changing the current context of the processor with a different one. This operation is performed by exploiting the interrupt mechanisms as following:

- Upon receiving the rollback signal, an interrupt request is issued to the processor.
- When executed, the interrupt routine overwrites the processor's stack with the content of the context memory, and then quits.
- Upon returning from the interrupt routine, the processor copies from the stack its context, and therefore resumes the execution from a state previously stored in the context memory, and considered as safe.

Both rollback and context saving mechanisms are based on synchronous interrupt requests. If any asynchronous interrupt occurs during the normal execution, the scenario will be the following:

TABLE I
RESOURCE OCCUPATION

Resource	Number	X-TMR XC2VP30	X-TMR XC2VP50
PowerPC	2	2	2
LUTs	5869	19725	19725
Block RAMs (16 KByte each)	15	15	15

- Both the processors capture the interrupt during the same execution cycles. Since both processors would execute the same interrupt routine, their contexts would be the same and thus the rollback approach works normally.
- The interrupt is captured in different execution cycles. The processors context would be different even if there is no SEU. The sanity check mechanism would signal a rollback operation and the two processors contexts will be restored to the one before the interrupt occurred.

In our design implementation we did not use other peripherals and we run very simple applications that do not need to management several interrupt request. However, in order to avoid to capture interrupts in different execution cycle and to have incoherent scenarios, it is possible to manage all the other interrupt requests at the end of each execution cycle. In this way, the synchronous handshake fashion of both processors will be preserved and the system will work properly. The main consequence of such solution is the delay in the interrupt management that can be estimated to be less of the time of one execution cycle.

III. Experimental Analysis

This section describes the experimental analysis we performed for assessing the capability of our approach in surviving soft errors affecting the processor's memory elements.

For this purpose we implemented a prototypical version of the proposed architecture, adopting a Virtex II-Pro XC2VP30 as the target device. This device embeds two PowerPC processors and sets available to designers a large logic fabric; it is therefore a good candidate for implementing the synchronized lockstep with rollback architecture. The resource occupation of the architecture is described in Table I. Please note that these resources have to be hardened too, since ionizing radiations may affect them, too. For this reason we adopted X-TMR tool from Xilinx that implements the TMR mitigation technique to all the design components with exception of the PowerPCs [11]. Thus, the proposed architecture is tolerant to the SEUs affecting the configuration memory. First we tested it on a Xilinx Virtex II-Pro XC3VP30, but the Xilinx tools are not able to complete the routing operation, because of the too high resources density. As consequence, we successfully implemented the design on a Xilinx Virtex II-Pro XC3VP50, which sets available more resources.

As shown in Table I, the resource occupation for the XC2VP30 and the XC2VP50 is about 336% greater than the implementation without X-TMR.

To assess the robustness of the proposed approach we performed extensive fault injection experiments by using the system presented in [10], which we recall in Section III-A.

TABLE II
CHARACTERISTICS OF THE CONSIDERED APPLICATIONS

Application	Execution time[1]	Code size	Data size	Injected	Wrong answers	Corrected	Silent	Checkpoint
	[Clock cycles]	[byte]	[byte]	[#]	[#]	[#]	[#]	[#]
M	626,580	100	576	10,000	248	1,790	7,962	13
F	25,046	76	128	10,000	323	5,392	4,285	33

[1] Please note that the Execution time is the time needed by both the processors to execute the application.

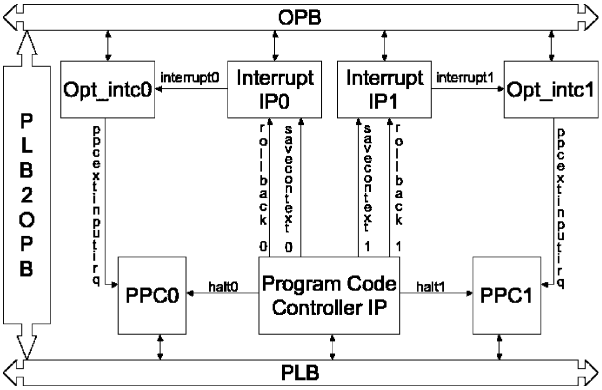

Fig. 3. Architecture of the synchronized lockstep with rollback approach.

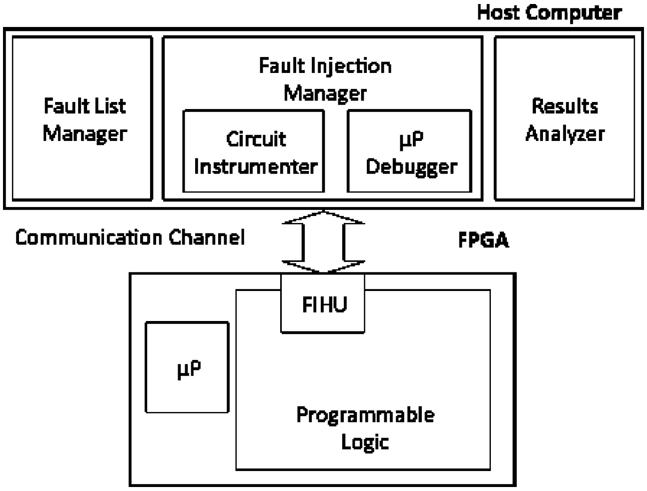

Fig. 4. Fault injection environment architecture.

Experimental results are described and commented in Section III-B, while Section III-C described future works.

A. The Fault Injection Environment

The architecture of the fault injection environment we used for gathering the experimental results is summarized in Fig. 4

The system is composed of the following main modules:

1. The *Fault List Manager* generates the fault list to be injected. Each fault is generated according to the assumption that each location has the same probability to be affected by a transient fault that happens at a random instant.
2. The *Fault Injection Hardware Unit (FIHU)* performs the fault injection process using part of the reconfigurable hardware. It manages the fault injection within the microprocessor internal resources. Moreover, it controls the observation of fault effects.
3. The *Fault Injection Manager* manages the fault-injection process controlling the FIHU and the μP debugger primitives.
4. The *Result Analyzer* analyzes the results and produces a report concerning the whole fault-injection campaign.

The communication channel between the host computer and the FIHU implemented within the FPGA uses the communication features provided by JTAG interface. The injection occurs during downtime of the device and the execution is restored with a fault that could manifest or that could be silent.

B. The Experimental Results

During our experiments we adopted the single bit flip in the processor's registers (user registers, special purpose registers, and control registers) as model for the soft errors that ionizing radiations may induce in the considered device. Moreover, we adopted two different applications as workloads for the processor:

- Matrix multiplication of two 12×12 matrices (M) computes the product of two 12×12 integer matrices.
- Finite impulse response filter (F) implements a filtering algorithm over a set of 32 samples.

The application code has not been modified with exception of a small prologue needed to register the interrupt routine. The characteristics of the considered applications are summarized in Table II (columns Execution time, Code size, and Data Size). The overhead due to X-TMR is not considered.

The execution time obtained depends on several factors as following:

- W: the time needed to execute the application in handshake. Considering the execution time of the application as T, the resulting time with the proposed architecture is $2T$.
- ΔP: the time needed to execute a small prologue for initialized the interrupt routine and to enable it in both processors. Such time depends on how many interrupts the users want to be recorded into the system. In the developed design we used just the interrupt needed for the context saving and the rollback operation.
- ΔIR: the time needed to execute the interrupt routine that implements either the context saving or the rollback. This routine takes about 200 instructions.
- ΔR: the time needed to execute a rollback operation. It can be defined as the time spanning in an execution cycle. For this reason it is not a fixed value but depends on the distance between two checkpoint.

It is worth to underline that ΔP can be considered as negligible when the execution time of the application is several times greater than the prologue's one. Furthermore, ΔIR is a fixed value that occurs every time a context saving or rollback has to be performed, thus the bigger the time distance between two checkpoint, the smaller the incidence on the system performance. Moreover, also ΔR becomes irrelevant for very long and complex application where the number of execution cycles is very high. The time analysis made does not consider the time for the handshake, because the Program Code Controller IP has been implemented in hardware and the resulting time is very small. As consequence, it can be considered irrelevant.

The effect of a fault in the processor's memory falls in one of the following categories:

- *Wrong answers* if the fault affects the processor in such a way that the outputs the application computes differ from the expected ones.
- *Corrected* if the fault is detected and removed from the system so that the application computes the expected results.
- *Silent* if the fault does not alter the application that computes the expected results, and does not trigger the error detection mechanisms the architecture embeds.

In our experiments we never experienced situations where a processor hangs (i.e., it never entered an endless loops in which write memory accesses, that triggers sanity check executions, are never performed). In case this eventuality has to be considered, the architecture has to be enriched with the adoption of time-out watchdog monitors.

From the fault injection results reported in Table II, the reader can observe that the architecture is able to survive most of the injected faults. For both applications about 97% of the injected faults fall either in the Silent or Corrected category. The number of Silent faults depends on the error propagation. In fact, the injected fault can be masked or latent (if it affects some registers never involved in any operation). The results also showed that a small fraction of the injected faults (about 3%) escapes the hardening strategy we implemented. We analyzed deeply such faults and we discovered that these are faults that are originated in the system in one execution cycle and that manifest themselves in the next clock cycle. As a result, during the execution cycle n the processor context is saved as safe, but it is faulty indeed, because of the latent error. If the fault manifests itself at execution cycle $n+i$ (with $i \geq 1$), the rollback will restore the processor context to the execution cycle $n+i-1$ that will always include the latent error and, for this reason, will be a faulty context. This analysis leads us to the extension of the proposed approach described in Section III-C.

C. Future Works

From the experimental results we gathered, we observed that few faults exist that may escape the rollback recovery with checkpoint mechanisms as they cross the boundary of the execution cycle. Such a fault corrupts the content of the processor registers during one execution cycle, but manifests itself only after the checkpoint operation is performed. As a result the stored context is faulty, and the rollback operation is no longer able to step back the system to a fault-free state.

To attack this scenario, we are evaluating an extension to our approach where multiple contexts are saved during program execution. Let us assume a memory is available storing a set $S = \{C_1, \ldots C_{n-1}, C_n\}$ of contexts, each saved after n successful sanity checks. In case the $n+1$ sanity check detects an error, the processor is step back through rollback to context C_n. In case the error belongs to the malicious category we discovered during our experiments, the repetition of sanity check $n+1$ will fail again as the fault affects context C_n and the execution cycle leading to sanity check $n+1$ is such that the fault arrives on the processors' outputs.

To solve this problem, we are modifying our architecture so that in case of multiple failing of sanity checks, rollback is performed using context C_{n-1}, and then C_{n-2}, and so forth. Thanks to this approach, we will be able to step back the processor to a context that is not affected by any fault, and therefore after a number of rollbacks it will eventually restart executing the program from a fault-free context.

This modification increases the complexity of the architecture, as well as the size of the memory storing the contexts. However, in case very high reliability is required, it seems a viable, although expensive, solution to mitigate fault spanning over multiple execution cycles.

In parallel to the aforementioned activity, we are planning to perform further validations of the architecture we devised. Accelerated radiation ground testing is needed for investigating on the effects of those faults that hit the processors in location non accessible through fault injection (i.e., the pipeline's registers), as well as the faults in the FPGA's fabric building the systems.

Moreover, additional fault models have to be considered:

- As the manufacturing technology continues to evolve, multiple bit upset have to be considered, too. It is therefore needed to understand which effects they may introduce in our architecture.
- As one pointed out by [8], critical situations may exist where one processor hangs. We have therefore to analyze this eventuality, and improve accordingly our architecture.

IV. Conclusion

As processors are deployed in safety- or mission-critical applications where low cost is a primary concern, new solutions to replace the traditional hardware redundancy approaches to fault tolerance are needed.

In this paper we proposed an implementation of rollback recovery using checkpoint that is suitable for hardening SRAM-based FPGA systems.

Despite the limitations of the current implementation of the approach, we were able to successfully mitigate the SEUs affecting a system built using two PowerPC processors, and we were able to recover from most of the SEUs provoking observable effects.

The experimental validation we performed also outlined some SEUs that escape the recovery mechanism we implemented. To cope with them we envisioned a scalable solution that is able to trade-off dependability with resource occupation.

References

[1] A. H. Johnston, "Radiation effects in advanced microelectronics technologies," *IEEE Trans. Nucl. Sci.*, vol. 45, no. 3, pt. 3, pp. 1339–1354, Jun. 1998.

[2] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE Trans. Rel.*, vol. 51, no. 2, pp. 111–122, Mar. 2002.

[3] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. S. Reorda, and M. Violante, "Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors," *IEEE Trans. Nucl. Sci.*, vol. 47, no. 6, pp. 2231–2236, Dec. 2000.

[4] E. Dupont, M. Nikolaidis, and P. Rohr, "Embedded robustness IPs for transient-error-free ICs," *IEEE Design Test Comput.*, vol. 19, no. 3, pp. 56–70, May/Jun. 2002.

[5] A. Mahmood and E. J. McCluckey, "Concurrent error detection using watchdog processors-A survey," *IEEE Trans. Comput.*, vol. 37, no. 2, pp. 160–174, Feb. 1988.

[6] P. Bernardi, L. Bolzani, M. Rebaudengo, M. S. Reorda, F. Vargas, and M. Violante, "Hybrid fault detection techniques in systems-on-a-chip," *IEEE Trans. Comput.*, vol. 55, no. 2, pp. 185–198, Feb. 2006.

[7] M. Pignol, "DMT and DT2: Two fault-tolerant architectures developed by CNES for COTS-based spacecraft supercomputers," in *Proc. IEEE Int. On-Line Testing Symp. 2006*, 2006, pp. 10–12.

[8] "PPC405 Lockstep System on ML310," Xilinx, Applicat. Note, XAPP564.

[9] D. K. Pradhan, *Fault-Tolerant Computer System Design*. Englewood Cliffs, NJ: Prentice-Hall, 1995.

[10] M. S. Reorda, L. Sterpone, M. Violante, M. Portela-Garcia, C. Lopez-Ongil, and L. Entrena, "Fault injection-based reliability evaluation of SoPCs," in *Proc. IEEE Eur. Test Symp.*, 2006, pp. 75–82.

[11] TMRTool. Xilinx, Inc., San Jose, CA. [Online]. Available: www.xilinx.com/esp/mil_aero/collateral/tmrtool_sellsheet_wr.pdf.