New Techniques for Improving the Performance of the Lockstep Architecture for SEEs Mitigation in FPGA Embedded Processors

F. Abate, L. Sterpone, C. A. Lisboa, L. Carro, and M. Violante

***Abstract*—The growing availability of embedded processors inside FPGAs provides unprecedented flexibility for system designers. The use of such devices for space or mission critical applications, however, is being delayed by the lack of effective low cost techniques to mitigate radiation induced errors. In this paper a non invasive approach for the implementation of fault tolerant systems based on COTS processors embedded in FPGAs, using lockstep in conjunction with checkpoint and rollback recovery, is presented. The proposed approach does not require modifications in the processor architecture or in the application software. The experimental validation of this approach through fault injection is described, the corresponding results are discussed, and the addition of a write history table as a means to reduce the performance overhead imposed by previous implementations is proposed and evaluated.**

***Index Terms*—Embedded processors reliability, single event effects, lockstep, checkpoint, rollback recovery, fault injection.**

I. Introduction

The evolution of the technology, providing ever smaller and faster devices, is at the same time bringing new challenges to the design of fault tolerant systems. The higher sensitiveness of those new devices to radiation induced transient faults, a long time concern for space and mission critical applications, makes the detection and correction of transient errors also a mandatory issue to be considered even in the design of general purpose systems used at sea level [1]. Moreover, the effects of single event transients (SETs) on combinational logic are now becoming a design concern as important as those of single event upsets (SEUs) affecting memory devices [2].

In parallel, the increasing availability of field programmable devices that include commercial off-the-shelf (COTS) processor cores makes this type of device the ideal platform for several applications. Their low cost and design flexibility are key factors to provide competitive products with shorter time to market, making them an ideal alternative for the consumer products industry. However, the effects of radiation on the internal components of such devices so far precluded their unrestricted use in most of space and mission critical applications.

In this class of devices, three different types of components must be protected against radiation: the configuration memory, used to define the function to be implemented by the reconfigurable logic, the reconfigurable logic itself, and the hardwired processor cores.

The protection of the configuration bits against SEUs can be achieved through the use of well known error detection and correction (EDAC) techniques [3]. More recently, the use of flash memories has been proposed as an alternative to EDAC. Besides providing lower power consumption, an important feature for space applications, flash memories are relatively immune to SEUs and SETs, due to the high amount of charge required to discharge the floating gate.

As to errors caused by SETs affecting the programmable logic components, they can be mitigated through the use of spatial redundancy techniques such as triple modular redundancy (TMR). While this approach implies a high penalty in terms of area and power consumption, it is so far the best available alternative for protection of the programmable logic inside SRAM-based FPGAs [3], [4].

In contrast, the mitigation of errors caused by radiation induced transient faults affecting the internal components of the embedded processor cores is still an open issue, undergoing intensive research. Despite the fact that the code and data used by the processors can be protected against radiation effects through the use of EDAC, after they are read and stored in the internal memory elements of the processor they are subject to corruption by radiation induced transients before they are used, leading to unpredictable results. Furthermore, even when fault tolerance techniques such as checkpoints are used to periodically save the system context for future recovery, this corrupted data can be inadvertently stored within the context, leading to latent errors that may manifest themselves later, when a recovery procedure requires the use of this information. Finally, when information used by the processor to manage the control flow is corrupted, catastrophic errors can occur, leading the system to irrecoverable states.

While several hardware and/or software based techniques for protection of the processor have been proposed in the literature, most of them cannot be applied for commercial off-the-shelf processors, for which the access to internal elements of the architecture is limited.

Manuscript received September 09, 2008; revised January 01, 2009. Current version published August 12, 2009. This work was supported in part by the Italian Ministry for University and Research (MIUR), and by the Alfa Nicron Project.

F. Abate, L. Sterpone, and M. Violante are with Politecnico di Torino, 10129 Torino, Italy (e-mail: massimo.violante@polito.it).

C. Lisboa and L. Carro are with Instituto de Informatica, PPGC, Universidade Federal do Rio Grande do Sul, CEP 91501-970, Porto Alegre, RS, Brazil.

Digital Object Identifier 10.1109/TNS.2009.2013237

0018-9499/$26.00 © 2009 IEEE

Software-based detection approaches work on faults affecting the control flow or data used by the program, and also provide coverage of those faults that affect the memory elements embedded in the processor, such as the processor's status word, or temporary registers used by the arithmetic and logic units [5], [6]. The main benefit stemming from software-based approaches is that fault detection is obtained only by modifying the software that runs on the processor, introducing instruction and information redundancies, and consistency checks among replicated computations. However, the increased dependability implies extra memory (for the additional data and instructions) and performance (due to the replicated computations and the consistency checks) overheads which may not be acceptable in some applications.

Hardware-based techniques insert redundant hardware in the system to make it more robust against single event effects (SEEs). One proposed approach is to attach special-purpose hardware modules known as watchdogs to the processor in order to monitor the control-flow execution, the data accesses patterns [7], and to perform consistency checks [8], 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 also at the software level, in order to decide the right amount of processing between each disarming of the watchdog. For this reason, 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 [9]. 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 behavior of the I-IP does not depend on the application the processor executes, and therefore it is widely portable among different applications.

Other researchers 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 [10], [11]. These approaches are particularly appealing in those cases where processor duplication does not impact severely the hardware cost. Moreover, since they do not require modifications to the software running on the duplicated processors, commercial off-the-shelf software components can be hardened seamlessly.

In the past, the use of checkpoints combined with rollback recovery as a means to build systems that can tolerate transient faults has also been proposed, and several studies aiming the implementation of architectures with this approach have been published. Among the proposed solutions, some require hardware support for its implementation, and some depend on software support, i.e., they imply modifications either in the hardware or in the software of the system to achieve fault tolerance. A comprehensive review of such studies can be found in [12].

This paper describes part of an ongoing research project aiming to build fault tolerant systems using COTS based FPGAs without the need to modify the processor's core architecture or the main application software. In this work, a new approach for the use of the lockstep mechanism [11] combined with checkpoints and rollback to resume the execution of the application from a safe state is proposed, in which the performance overhead imposed by previous solutions is significantly reduced. The trade-offs between the frequency of checkpoints, the fault detection latency and the error correction time are discussed, and the use of an IP module to speed up checkpoints for applications with large data segments is proposed and evaluated.

The remainder of the paper is organized as follows: Section II reviews the basic concepts applied in the proposed implementation, and Section III describes the details of the resulting architecture. Section IV describes the fault injection experiments conducted to validate the implementation and discusses their results, while in Section V the modifications aiming to improve the overall performance of the application are presented and discussed. Finally, Section VI summarizes the conclusions and points to future work to be developed in the scope of this research project.

II. Lockstep, Checkpoint, and Rollback Recovery

Aiming at detecting errors affecting the operation of the processor, the *lockstep* technique uses two identical processors running in parallel the same application. The processors are first synchronized to start from the same state and both receive the same inputs, and therefore the states of the two processors should be equal at every clock cycle, unless an abnormal condition occurs. This characteristic of lockstep allows for the detection of errors affecting one of the processors through the periodical comparison of the processors' states. The retrieval and comparison of processor states, here named *consistency check*, is performed after the program has been executed for a predefined amount of time or whenever a milestone is reached during program execution (e.g., a value is ready for being committed to the program user or for being written in memory). When the states differ, the execution of the application must be interrupted, and the processors must restart the computation from a previous error-free state.

To restart the application from its beginning is very expensive in terms of computation time, and sometimes is also not feasible. In order to avoid that, *checkpoints* are used in conjunction with lockstep to keep a copy of the last error-free state in a safe storage. With this purpose, whenever a consistency check shows that the states of the processors are equal, a copy of all information required to restore the processors to that state when an error is detected is saved in a storage device which is protected against soft errors or that allows the detection and correction of those errors when they occur. This set of information is usually named *context*, and encompasses all information required to univocally define the state of the processor-based system (it can include the contents of the processor's registers, the program counter, the cache, the main memory, etc.).

If the consistency check fails, i.e., the states of the two processors are different, an operation named *rollback* must be performed to return both processors to a previous error-free state.

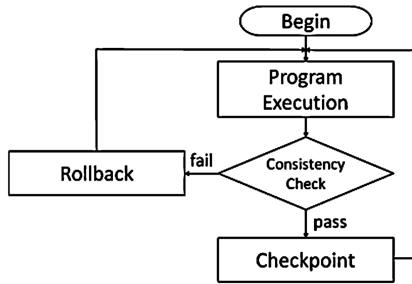

Fig. 1. Flow chart of rollback recovery using checkpoint.

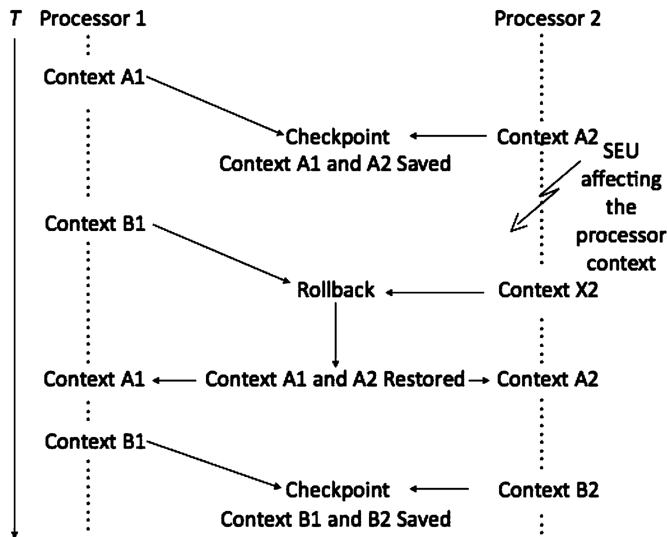

Fig. 2. Example of execution of rollback recovery using checkpoint.

This is done by retrieving the most recent context saved during a previous checkpoint and using it to restore the processors to that state, from which the execution of the application is resumed.

The flowchart of the above described technique is depicted in Fig. 1. When a rollback is performed, the computation executed since the last checkpoint until the moment when the consistency check was executed must be repeated.

Fig. 2 shows an example of application execution flow using the lockstep technique combined with checkpoint and rollback recovery. The arrow on the left indicates the timeline (T).

Initially, processor 1 executes one portion of the application until it reaches a predefined point. The context of processor 1 at this point is A1. Then, processor 2 executes the same portion of the application, reaching the same point with context A2. When both processors reached the same predefined point, their contexts are compared and, if they are equal, a checkpoint is performed, saving the states of the two processors in a soft error tolerant memory.

Next, the execution of the application is resumed, with processor 1 performing another portion of the code until it reaches a second predefined point, with context B1, and then processor 2 executes the same portion of the application, stopping at the same second predefined point, with context B2. At this point a new consistency check is done and, if no error occurred, a new checkpoint is performed, saving contexts B1 and B2, and so on, until the whole application has been successfully executed by both processors.

Now, let us suppose that, as shown in Fig. 2, one SEU occurs and causes one error while processor 2 is processing the second portion of the application code. In this case, when it reaches the second predefined point and the consistency check is performed, the state of processor 2 is X2, instead of B2, which indicates that one error occurred and that, as a consequence, a rollback must be performed.

The rollback operation, then, restores both processors to their last error-free states using the information saved during the last checkpoint performed by the system, i.e., contexts A1 and A2, respectively. The execution of the application is then resumed as previously described, with processor 1 and then processor 2 executing, one at a time, the same portion of the application that was affected by the error, and if no other error occurs the processors finally reach the correct states B1 and B2 and a new consistency check is performed, saving contexts B1 and B2. This way, the error caused by the SEU has been detected during the consistency check, and corrected by the repeated execution of the code segment in which the error has occurred.

While the techniques used in this approach are apparently simple, their implementation is not trivial, demanding the careful consideration of several issues.

A particularly critical aspect is the criteria to be used when defining at which points the application should be interrupted and a consistency check performed, since it can severely impact the performance of the system, the error detection latency, as well as the time required to recover from an erroneous state. Clearly, checking and saving the states of both processors at every cycle of execution provides the shortest fault detection and error recovery times. However, this imposes unacceptable performance penalties to any application. In contrast, long intervals between consecutive checkpoints may lead to catastrophic consequences due to the error propagation in systems where the results produced by one module are forwarded to other modules for further processing, as well as to the loss of deadlines in real-time applications when one error occurs. Therefore, a suitable trade-off between the frequency of checkpoints, error detection latency and recovery time must be established, according to the characteristics of the application, and taking into account the implementation cost of the consistency check as well.

A second issue is the definition of the consistency check procedure to be adopted. Considering that the consistency check aims to detect the occurrence of faults affecting the correct operation of the system, the consistency check method plays an important role in the achievement of the fault tolerance capabilities of the system. The optimal balance between maximum fault detection capability and minimum consistency check implementation cost must be pursued.

In the definition of the context of the processors, designers must identify the minimum set of information that is necessary to allow the system to be restored to an error-free state when a fault is detected. The amount of data to be saved affects the time required to perform checkpoints and also to rollback when one error is detected. Therefore, in order to provide lower performance overhead during normal operation, as well as faster recovery when an error occurs, the minimum transfer time for

those operations must be pursued, together with a low implementation cost.

The storage device used to save the context data must be immune to the type of faults that the system tolerates, in order to ensure that the information used to restore the processors to a previous state when one error is detected has been also preserved from such faults between the checkpoint and rollback operations.

Finally, the most efficient methods should be used to develop the checkpoint and rollback procedures, since they require access to all the memory elements containing the context of the processors, and have to be performed every time a checkpoint must be stored, after a successful consistency check, or a rollback must be performed to load an error-free context into the processors, when one error is detected by the consistency check. Depending on the definition of the context, the frequency of consistency check execution, as well as the error rate, checkpoint/rollback operations may be performed very frequently, and therefore the time spent while moving data to and from the processor must be minimized.

III. The Proposed Implementation

The implementation of synchronized lockstep combined with checkpoints and rollback recovery presented in this paper was inspired in the approaches proposed in [10] and [11], and it is an extension of the implementation presented in [13]. It has been conceived to harden processor cores embedded in FPGA devices against soft errors affecting the internal memory elements of the processors, and has been initially implemented using a Xilinx Virtex II Pro FPGA, which embeds two 32-bit IBM Power PC 405 hard processor cores. However, the approach is general and it can be extended to different FPGA devices with two embedded processors (e.g., the Actel devices with embedded ARM processors).

In the following subsections, we describe the adopted solutions for the aforementioned main issues.

A. Consistency Check Implementation

Due to the availability of two processor cores in the devices used for the implementation, processor duplication with output comparison was adopted to implement the consistency check. The developed approach uses two processors running the same application software. Considering that the processors are synchronized, and executing the very same software, they are expected to perform exactly the same operations. By observing the information travelling to and from the processor it is therefore possible to identify fault-induced misbehaviors.

The consistency check is performed every time the two processors perform a write cycle, i.e., every time they send information to the memory. The control bus is monitored to detect when each processor is issuing a write operation. 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 exactly the same segment of the application. As soon as the second processor executed the write operation, it is also stopped, and the consistency check is performed by comparing the information sent through the data and address busses by each processor

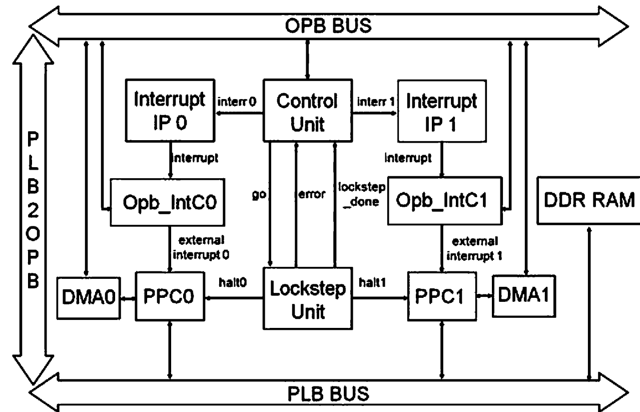

Fig. 3. Architecture of the synchronized lockstep with rollback.

in order to confirm that both wrote the same data in the same address. After a successful consistency check, a checkpoint is performed and the first processor resumes the execution of the software.

The need to stop one processor while the other is running the application arises from the fact that the device used in this work has a single memory, which is shared by both processors through the PLB bus, as shown in Fig. 3. Therefore, only one processor can access the memory at each time. To overcome this restriction, in a previous work targeting the same device [13] both processors run in parallel, but only one of them writes the results of the computation into memory. In that work, however, when a mismatch occurs the system cannot know which of the processors failed, and therefore the technique proposed there has no error correction capability, being only able to detect errors, while the technique proposed here uses checkpoints to allow error correction.

The frequency of checkpoints can affect both the performance and the dependability of the implemented solution. For the analysis of those parameters, we define the time spanning between two consecutive checkpoints as *execution cycle*, while we define *lockstep cycle* as the time spanning between the start of the execution of one application segment by the first processor, and the completion of the write operation by the second processor.

In our approach, one execution cycle can include one or more lockstep cycles, and only at the end of each execution cycle a dedicated hardware module performs the consistency check and triggers the checkpoint operation to save the status of the memory elements of both processors in a dedicated memory area, thereby saving the context of the system.

B. Context Definition and Storage

In this work the context to be saved during the checkpoint operation includes the contents of the 43 user registers (32 general purpose registers and 11 special purpose registers, 32-bit wide), program counter, stack pointer, processor status word, and the data segment of each processor. It does not include the status of the processor's cache, which therefore is assumed to be disabled. However, the implementation can be extended to deal with the cache too, by flushing the data cache contents to the main memory during checkpoint, before the context is saved, and by invalidating the data/instruction cache upon execution of a rollback operation.

For the sake of this paper we assume the memory used to store the processor's context is immune to SEUs, i.e., it is hardened using suitable EDAC codes as well as memory scrubbing.

C. Overall Architecture

The architecture of the proposed implementation is shown in Fig. 3, and it includes the following modules:

- *PPC0* and *PPC1*: the two Power PC 405 processors embedded in the FPGA, working in lockstep mode.
- *Interrupt IP0* and *Interrupt IP1*: two custom IP modules used to trigger the interrupt routines that perform the checkpoint and rollback operations for each processor.
- *Opb_intC0* and *Opb_intC1*: interrupt controller IPs provided by Xilinx that are connected to PPC0 and PPC1 to manage the interrupt requests from Interrupt IPs 0 and 1, sending the interrupts signals to the processors.
- *DMA0* and *DMA1*: DMA controller IPs provided by Xilinx, used to provide faster transfer of context information between the application data segments and the context saving storage during checkpoints.
- *Lockstep Unit*: a custom IP that monitors the operations of the two processors, using the `halt0`/`halt1` signals to stop each processor immediately after it issues a write operation, and restart them to resume execution. The bus master signal is used to determine which processor is currently writing to memory. Whenever it receives the `go` signal from the Control Unit, the Lockstep Unit starts one lockstep cycle. Once both processors have performed the same write instruction, it performs the consistency check and uses the `lockstep_done` signal to inform to the Control Unit that a cycle has been completed, and also activates the `error` signal when a mismatch occurs.
- *Control Unit*: a custom IP that interacts with the Lockstep Unit to execute the application in lockstep mode and receives the results of the consistency checks. After the predefined number of successful write operations has been performed, it triggers the interrupt routines on each processor to perform the checkpoints when no error occurred. When a mismatch has been found, the interrupt routines perform a rollback operation. This new approach represents a major change with respect to [13], providing improvements in terms of dependability and performance.

D. Implementation Improvements and Details

The system includes a standard DDR RAM memory for both code and data segments storage, which is divided into two independent addressing spaces, each used by only one processor, i.e., one processor cannot read from nor write into the addressing space of the other. The context of each processor and the copies of their data segments are also stored in DRAM, in areas not used by the application software.

In order to minimize the time needed for checkpoint and rollback execution, they have been implemented using the interrupt mechanisms made available by the processors. When an interrupt request is received the processor stops executing the application, saves its context into the stack, and starts executing the corresponding interrupt handling routine. When the interrupt handling routine ends, the processor restores its context from the stack and resumes the execution of the application from the point it has been interrupted.

During checkpoint the system performs the following steps:

- After the interrupt routine request is raised, the processor saves its context in the stack.
- The checkpoint interrupt service routine saves the contents of the stack in the *context memory*.
- The checkpoint interrupt service routine copies the section of the main memory where the program's data segment is stored to the *context memory*. This operation is performed using the DMA controller for a direct memory-to-memory data transfer.

Conversely, the rollback mechanism restores a previously saved context, performing the following operations:

- The rollback interrupt routine copies the previously saved processor's context from the context memory to the stack.
- The rollback interrupt routine uses a DMA transfer to copy the stored data from the context memory to the program's data segment.
- When the processor returns from the rollback interrupt routine, it overwrites the processor's context with the stack contents, thus resuming program execution from the same error-free state saved during the last checkpoint.

The above described implementation of the rollback and checkpoint operations brings significant improvements with respect to the one described in [13], which requires tailoring the application to be run in the system. Specifically, in that implementation the data segment contents were not saved in the context, which required the application to be written in a particular way in order to preserve the integrity of the data between a given checkpoint and a possible rollback following it. The program could only write new values to variables in memory at the end of the execution, otherwise a rollback performed in the middle of the execution could lead the processor to an inconsistent state. In such cases, the context information would be reversed to a safe state, while memory variables would remain with their last, possibly erroneous, contents. That restriction imposed a strong limitation for application developers.

Moreover, in the approach presented here consistency checks are executed every time a write occurs, while checkpoints are triggered only after the number of write operations defined at design time has been performed. This brings two new important improvements with respect to [13]. The first one is the reduction of the performance overhead, since the checkpoint operation implies saving the entire register set and data segment contents of both processors into memory. The second advantage regards the dependability of the solution. In fact, the experimental analysis described in [13] showed that in some cases SEEs may remain latent in a context, i.e., one SEE is latched by one of the processors (e.g., in a general register) during execution cycle n, but the affected data is used for computation only during execution cycle $n + x$. In such cases, a faulty context is saved during the checkpoint following execution cycle n, thus preventing the successful execution of the recovery mechanism after the error is detected by the consistency check during any subsequent execution cycle, and leading the system to an endless sequence of rollback operations. By extending the execution cycle to include more write operations, the probability that a latent SEE

TABLE I
SENSITIVE BITS FOR IP

Resource	Sensitive Bits
Control Unit	41,789
Lockstep Unit	59,173
Interrupt IP 0 and 1	2 × 89,421
Opb_intC 0 and 1	2 × 4,595
DMA0 and 1	2 × 25,744
Glue logic	75,338
TOTAL	415,820

manifests itself within the same execution cycle during which it is latched has been increased, and so the probability of successful execution of the rollback, thereby providing higher dependability for the whole system.

IV. Fault Tolerance Analysis

This section describes and discusses the fault injection experiments that have been performed for assessing the capability of the proposed approach to cope with soft errors affecting the processor's memory elements. In this phase of the research work, only SEEs affecting the processor's registers have been investigated. However, the use of ground facilities to explore other types of radiation induced effects is planned as future work.

Besides the two PowerPC processors, the proposed architecture uses only a limited amount of the FPGA resources: 6 991 slices and 48 16-kB blocks of RAM. Therefore, it is suitable for being embedded in complex designs, where larger devices are expected to be used.

Concerning the FPGA's configuration memory, the number of bits that may cause a system failure has been computed using the STAR tool [14]. Table I reports the obtained results, which have been classified according to the different modules of the architecture, plus the glue logic implementing the processor chipset, e.g., to interface with the DDR RAM. Since the configuration memory of the selected device is composed by 11 589 920 bits, one can see that only 3.6% of them are expected to be sensitive.

While the present work proposes a technique to cope with errors affecting only the processor cores embedded in the FPGA, it is important to note that the configuration memory and the reconfigurable logic themselves must be hardened too, since ionizing radiations may also affect them. However, within the scope of this work, the proposed architecture has been deemed tolerant to the SEUs affecting the configuration memory and the reconfigurable logic, and no faults have been injected in those elements.

For the specific devices used to implement and test the technique proposed in this work, the protection of the configuration memory and the reconfigurable logic could be implemented through the use of the X-TMR tool from Xilinx, which uses the triple modular redundancy (TMR) technique to harden all the design components against SETs, with exception of the Power PCs [4].

However, TMR is not a bullet proof technique, since it uses a voter circuit to choose, among the outputs of three modules, which are the correct ones. Although the area of the voter circuit is usually much smaller than that of the tripled modules that it protects, its components are still subject to radiation effects and must also be hardened by suitable techniques. Among those, the use of larger transistors dimensions and the use of one additional TMR instance to triple the voter circuit and then use a fourth voter to choose the correct output are the more widely used to minimize the error rate. Furthermore, in the unlikely event of two simultaneous faults affecting the same output bit of two of the tripled modules, the voter will silently choose the wrong result as the one to be forwarded to the output of the circuit, with catastrophic consequences. A deeper discussion of TMR hardening techniques, however, is out of the scope of this paper.

The injection of faults in the internal registers of the PPC microprocessors has been performed using the method described in [15]. To simulate the occurrence of a Single Event Upset (SEU), during each run of the application one bit of one internal register of the microprocessor is complemented. The register and the bit to be flipped are selected randomly, using a specially developed hardware. A *Fault Injection Hardware Unit* (*FIHU*), placed between a host computer and the microprocessors, performs the fault injection process using part of the reconfigurable hardware and manages the injection of faults affecting the microprocessors internal elements. On the host computer, a *Fault Injection Manager* controls the fault-injection process through the FIHU and using the μP debugger primitives. Detailed reports concerning the results obtained during the fault-injection campaign are produced by a *Result Analyzer* module. The communication channel between the host computer and the FIHU implemented within the FPGA exploits the communication features provided by the JTAG interface.

For analysis purposes, the effects of the fault injection on the outputs of the system have been classified as follows:

- *wrong answers*, when the outputs of both processors were equal, but different from the expected ones;
- *corrected*, when the error caused by the injected fault was detected and corrected by the implemented mechanism, so that the output results were the same for both processors, and were equal to the expected ones;
- *latent*, when the injected fault caused a latent error which escaped the detection and correction mechanism embedded in the system, and therefore after the execution of rollback and repetition of the computation the outputs produced by the two processors were still different; and
- *silent*, when the injected fault did not have any consequence on the results generated by the application.

A preliminary set of results has been collected using as benchmark application the multiplication of two 3×3 integer matrices. The application code has not been modified, except for the insertion before it of a small prologue needed to register the interrupt routine. The application has a code length of 100 bytes and requires 1 922 272 clock cycles for completion. For the selected application we analyzed the overhead introduced by checkpoint execution, and the sensitivity of the hardened system to SEEs.

The application has been executed with three different versions of the system, which performed a checkpoint at every cycle (saving 100% of the contexts), at every 3 cycles (33% of contexts) and at every 6 cycles (16.7% of contexts), respectively, and the collected results are reported in Table II.

These figures confirm that the execution of one checkpoint after each write instruction imposes a too heavy penalty on the

TABLE II
RESULTS OF FAULT INJECTION ON THE PROCESSORS

Context Savings [%]	Execution time [Clock cycles]	Code size [bytes]	Data size [bytes]	Injected [#]	Wrong Answer [#]	Corrected [#]	Latent [#]	Silent [#]
100.0	17,219,076	100	36	1,800	0	200	116	1,484
33.0	14,049,761	100	36	1,800	0	321	87	1,392
16.7	11,216,452	100	36	1,800	0	440	29	1,331

performance of the system, while limiting the checkpoints to one at every 6 writes leads to a much lower overhead. As far as SEE sensitivity is concerned, one can notice that all the injected faults have been appropriately handled in our experiments. The faults that had effects on the program execution have been corrected thanks to rollback and those that caused latent errors have been detected at the end of the computation, since the data segments of the two processors contained results that were different among them. Moreover, it is worth noticing that the number of latent errors decreases sharply when the frequency of context savings decreases, while the number of corrected errors increases. This fact shows that errors are less likely to remain latent at the end of the execution cycle when a larger number of writes per execution cycle is used.

The preliminary experimental analysis confirmed that the proposed approach is an efficient and scalable method for hardening processors systems when two processors are available at low cost. However, it has also shown that some errors may become latent and not be detected by the proposed mechanism at the end of the execution cycle in which they have been latched. To cope with this type of errors, a scalable solution, able to trade-off dependability with resource occupation, has been devised.

This solution extends our technique by saving multiple consecutive contexts during the execution of the application. This way, when one error is detected during the consistency check performed after a given execution cycle, a rollback to the context saved during the last checkpoint is performed, and the execution of the application is resumed from that point. If the detected error was a latent one, the consistency check will fail again at the end of the same execution cycle, since the erroneous data was saved during the last checkpoint. This shows that the last saved context is not error-free, and so the system performs two consecutive rollbacks, to bring the system to the last but one saved state, and resumes the execution from there. If the latent error was latched only during the last checkpoint, this will lead the system back to an error-free state and the execution of the application will proceed normally. Otherwise, the system will now perform three consecutive rollbacks, and so on, until it reaches a context not affected by the latent error and recovers from it, or the context buffer is exhausted.

While this extension implies higher costs, due to the need of a larger memory to store contexts, its application can be scaled according to the criticality of the application to be protected, being a viable solution to cope with latent errors in the proposed system.

V. ADDITIONAL PERFORMANCE IMPROVEMENT EXPERIMENTS

The implementation described in Section III improved the performance and the dependability of the system by reducing the number of checkpoints performed during the execution of the application. In the experiments described in Section IV the number of lockstep cycles per execution cycle has been changed from 1 to 3 and 6, using an application with a very small data segment, which performs the multiplication of 3×3 matrices. However, considering that the whole data segments of the applications running in both processors must be saved during checkpoints, there is still a significant performance penalty for applications with large data segments. Aiming to further improve the performance of the system for this kind of application, another experiment has been conducted, in which one additional IP, named Write History Table (WHT), has been included in the system, as shown in Fig. 4.

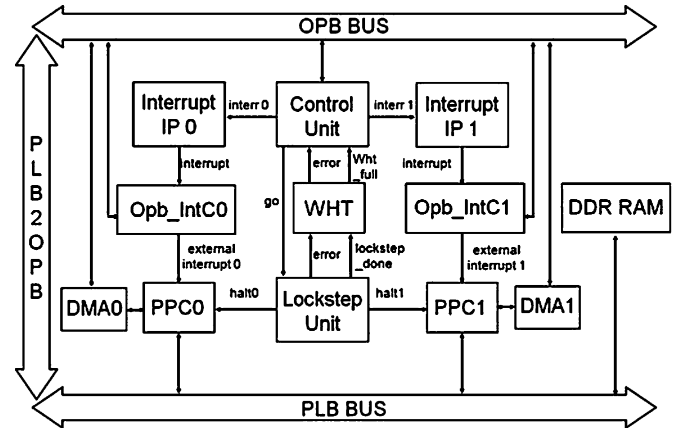

Fig. 4. Architecture modified to include the WHT.

The WHT has been inserted between the Lockstep Unit and the Control Unit, and it is used to temporarily store the addresses and values that have been written by the application during one execution cycle. Whenever the consistency check performed by the Lockstep Unit determines that address and value are consistent between both processors, they are stored in a new entry of the table inside WHT. When the table is full, the WHT IP sends the `wht_full` signal to the Control Unit, which then performs a checkpoint. When the Lockstep Unit detects an error, the address-value pairs already stored in the WHT are flushed and the error signal is passed forward to the Control Unit, which then requests a rollback operation.

Considering that the consistency checks ensure that the data written by both processors is the same, now only one copy of the data segment is kept in the so-called *data segment mirror area*. Moreover, the checkpoint operation performed by the interrupt service routine has been modified in order to write into the data segment mirror area only those words which have been changed by the application after the last checkpoint, thereby

avoiding transferring the whole data segment of both processors to memory, which can demand a long time for applications with large data segments. In order to accomplish this task, during checkpoints data is now copied from the WHT to the data segment mirror area using processor instructions, and no longer DMA transfers.

The rollback operation, in turn, besides restoring the processor contexts to the stack area of each application, as before, now copies the single data segment mirror area to the data segments of both processors using DMA transfers.

In order to confirm that these modifications bring better performance for applications with large data segments, the matrix multiplication application has been performed several times, with matrix sizes varying from 2×2 to 20×20, and the number of cycles required to execute the whole application, including all checkpoints, has been measured for different configurations of WHT, respectively with 8, 16, and 31 entries. This means that the number of lockstep cycles per execution cycle has been also increased when compared with the previous experiments, with one checkpoint being performed after every 8, 16, or 31 write operations, respectively. As show in the previous section, this is also a dependability increasing factor.

To allow comparing the impact on performance, the same applications have also been run on the previous version of the system (without WHT), using the same number of lockstep cycles per execution cycle (8, 16, and 31), and the average number of cycles per write operation has been calculated.

The graphics in Fig. 5 show the comparison of the average number of cycles per write operation required by each implementation for each quantity of lockstep cycles per execution cycle. In those figures, Lockstep Only refers to the implementation described in Section III, while Lockstep with WHT refers to the one described in this section.

By analyzing the results, the expected improvement of performance provided by the introduction of the WHT has been confirmed for applications with larger data segments. For the implementation of lockstep described in Section III (dotted lines), as the size of the data segment increases the average number of cycles per write operation grows almost linearly. In contrast, for the system with WHT (solid lines) the number of cycles remains almost constant after a certain data segment size is reached.

In the analysis of the graphics, it is important to highlight that for applications with small data segments, in this experiment represented by multiplication of small matrices, the use of WHT does not improve the performance. Also, the break-even point, i.e., the size of the data segment from which the use of WHT becomes an advantage, increases with the number of lockstep cycles per execution cycle (which is the same as the number of entries in the WHT). This is due to the use of DMA transfers to save the data during checkpoints in the system without WHT, since for small data segments the DMA memory-to-memory transfer is faster than the execution of 8, 16, or 31 transfers from the WHT slave registers to memory using processor instructions.

Table III shows the relationship between the quantity of entries in the WHT (each entry holds one address-value pair) and the size of the data segment of the applications in bytes, for the points where the use of WHT becomes advantageous.

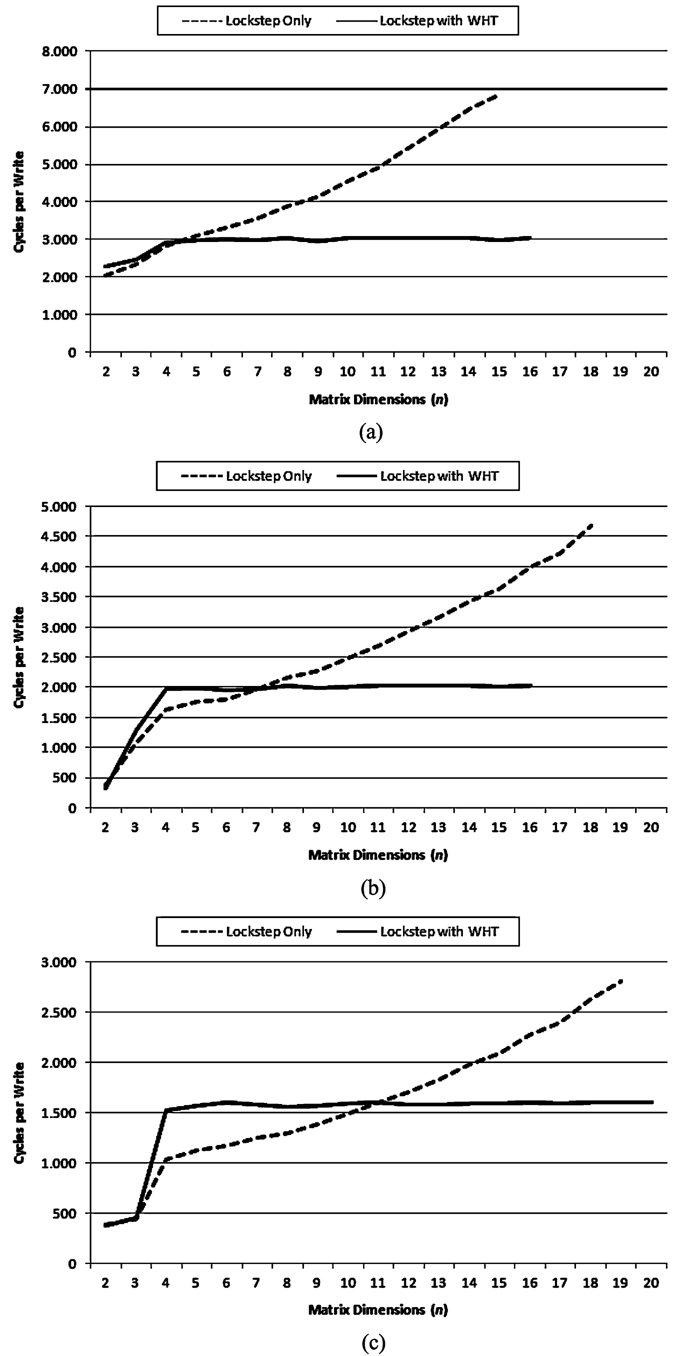

Fig. 5. Average cycles per write vs. matrix size comparison. (a) Checkpoints at every 8 writes, (b) Checkpoints at every 16 writes, (c) Checkpoints at every 31 writes.

TABLE III
DATA SEGMENT SIZE BREAK-EVEN POINT FOR USE OF WHT

WHT size (# of entries)	8	16	31
Matrix dimensions	5×5	7×7	11×11
Data segment size (bytes)	100	196	484

Through those experiments, it has been shown that the use of the WHT IP can indeed improve the performance of applications with large data segments, and that the number of entries in the WHT can be adjusted at design time in order to obtain the best results for a given data segment size.

As to the fault tolerance capabilities of the system with WHT, they are the same described in Section IV, since the same assumptions concerning the use of memory protected by EDAC techniques and use of TMR to protect the reconfigurable logic inside the FPGA have been used. The dependability of the system, however, increases with the higher number of lockstep cycles per execution cycle adopted in the implementation described in this section.

VI. Conclusion and Future Works

The use of SRAM-based FPGAs with embedded processors for the implementation of safety- or mission-critical systems has been precluded so far by the lack of appropriate techniques to cope with radiation induced errors affecting the internal elements of the processors. The increasing availability of FPGAs with multiple embedded COTS processors makes feasible the development of new low cost techniques to implement fault tolerance without modification of the hardware and/or of the software running on the processors.

In this paper, two new incremental approaches for the implementation of systems tolerant to radiation induced faults, using the lockstep technique combined with checkpoints and rollback recovery, have been proposed.

The first one included the application data segment in memory as part of the context to be saved during checkpoints, and also increased the number of write cycles between checkpoints, thereby precluding the need for special programming rules to be followed during the development of the application software, and reducing the number of non detected latent faults. The second approach introduced an additional IP module, named Write History Table, aiming to reduce the time required to perform checkpoints. This was accomplished by writing to the data segment mirror area only those memory words which have been modified since the last checkpoint.

By reducing the number of checkpoints, as well as the amount of data to be stored during each checkpoint, the proposed improvements allow to decrease the time dedicated to checkpoints, thereby imposing less performance overhead to the application, when compared to previously proposed approaches. At the same time, the reduction of latent faults obtained by increasing the number of write operations per execution cycle, leads to improved system dependability provided by the reduction of latent errors. All those benefits are provided without requiring any modification in the architecture of the embedded processors or in the main application software running on them.

Further investigations are under development, namely: analysis of performance degradation due to rollback execution and the use of a context addressable table to implement WHT, in order to keep in the table only the last value written into a given address. In parallel, further validations of the architecture are being planned, including accelerated radiation ground testing for investigating the effects of faults that hit the processors in locations not reachable through simulated fault injection, such as the processors' pipeline registers, as well as the use of additional fault models in the experiments, such as multiple bit upsets. It is expected that the radiation experiments results will report other types of errors due to the propagation of SEE in the logic, such as Single Event Transients (SETs) and Multiple Bit Upsets (MBUs).

References

[1] T. Heijmen, Radiation Induced Soft Errors in Digital Circuits: A Literature Survey Philips Electronics Natl. Lab., Netherlands, Report 2002/828, Aug. 2002.

[2] R. Baumann, "Soft Errors in Advanced Computer Systems," in *IEEE Design and Test of Computers*. New-York-London: IEEE Computer Society, May–June 2005, vol. 22, pp. 258–266, 3.

[3] B. W. Johnson, *Design and Analysis of Fault Tolerant Digital Systems: Solutions Manual*. Reading, MA: Addison-Wesley Publishing Company, Oct. 1994.

[4] [Online]. Available: www.xilinx.com/esp/mil_aero/collateral/tmrtool_sellsheet_wr.pdf

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

[6] 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.

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

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

[9] 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. Computers*, vol. 55, no. 2, pp. 185–198, Feb. 2006.

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

[11] PPC405 Lockstep system on ML310 Xilinx App. Note, XAPP564.

[12] D. K. Pradhan, *Fault-Tolerant Computer System Design*. : Prentice-Hall, 1995.

[13] F. Abate, L. Sterpone, and M. Violante, "A new mitigation approach for soft errors in embedded processors," *IEEE Trans. Nucl. Sci.*, vol. 55, no. 4, pt. 1, pp. 2063–2069, Aug. 2008, 2008.

[14] L. Sterpone and M. Violante, "A new analytical approach to estimate the effects of SEUs in TMR architectures implemented through SRAM-based FPGAs," *IEEE Trans. Nucl. Sci.*, vol. 52, no. 6, pp. 2217–2223, Dec. 2005, 2005.

[15] 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.