A Low-Cost SEE Mitigation Solution for Soft-Processors Embedded in Systems On Programmable Chips

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

C. Meinhardt, R. Reis
Unviersidade Federal do Rio Grande do Sul,
Porto Alegre, Brasil

Abstract

The availability of multimillion Commercial-Off-The-Shelf (COTS) Field Programmable Gate Arrays (FPGAs) is making now possible the implementation on a single device of complex systems embedding processor cores as well as huge memories and ad-hoc hardware accelerators exploiting the programmable logic (Systems on Programmable Chip, or SoPCs). When deployed in safety- or mission-critical applications, as avionic- and space-oriented ones, Singe Event Effects (SEEs) affecting COTS FPGA, which may have catastrophic effects if neglected, have to be considered and SEE mitigation techniques have to be employed. In this paper we explore the adoption of known techniques (such as lockstep, checkpointing and rollback recovery) for SEE mitigation to processors cores embedded in SoPCs, and propose their customization, specifically addressing the characteristics of programmable devices. Since the resulting design flow can easily be supported by automation tools, its adoption is particularly suitable to reduce the design and validation costs. Experimental results show the effectiveness of the proposed approach when compared to conventional TMR-based solutions.

1.Introduction

New fabrication technologies are today allowing manufacturers of Field Programmable Gate Arrays (FPGAs) to integrate unprecedented amounts of resources in their newest devices. Recent Commercial-Off-The-Shelf (COTS) devices, intended for mainstream general-purpose applications, can offer to designers more than 600K logic cells, and about 20 Mbits of embedded memory. Developers of avionic- or space-oriented applications can benefits from the availability of such devices to integrate on single component very complex systems, obtaining the so-called Systems on Programmable Chips (SoPCs), embedding processors cores, high-speed memories and ad-hoc hardware accelerators all tightly interconnected by high-speed buses. At the same time, the relatively high sensitivity of COTS FPGAs to the environment where avionic or space electronic systems operate makes the adoption of error mitigation techniques mandatory.

As far as error mitigation techniques are considered, the adoption of fault tolerant FPGA technologies is not considered as a generally viable approach, as these components are too limited in terms of resource availability. Approaches have been proposed, for example in [1][2], to enable the adoption of COTS FPGAs in radioactive environments, which exploit hardware and/or time redundancies to provide the needed level of error tolerance at the cost of high resource overhead. Most of the area overhead comes from the mitigation of the processors that SoPCs embed. Indeed, advanced microprocessors are today used in complex SoPCs, which account for a not-negligible share of the resources the selected FPGA may offer. As an example, if we consider a SoPC exploiting a Sparc-like processor implemented using the largest Xilinx Virtex 4 device now available, the xc4vlx200, and designed according to the Triple Modular Redundancy (TMR) approach to survive radiation-induced errors, we have that about 40% of the available resources must be devoted to the processor.

The main contribution of this paper consists in exploring new hardening solutions that are specific of SoPCs, although being based on traditional techniques. We propose a design flow to that can effectively be adopted by designers in mitigating radiation-induced errors affecting processor cores embedded in FPGA-based SoPCs. The flow exploits known concepts as lockstep, checkpoint and rollback recovery [3][4] to provide a balance between resource overhead (that is lower than in the TMR case) and fault tolerance. A major advantage of the flow is that it can be easily automated, thus greatly reducing the total development costs, while increasing the quality of the resulting product.

We developed a prototypical implementation of a design environment supporting the proposed flow, and we applied it to a complex processor core. The experimental evaluation of the obtained design showed the effectiveness and limitations of the proposed flow.

The rest of the paper is organized as follows. Section 2 introduces the problem. Section 3 and 4 present the assumptions we made and the design flow we propose. Section 5 analyzes the results we gathered on a case study. Finally, section 6 draws some conclusions.

978-3-9810801-5-5/DATE09 © 2009 EDAA

2.Problem statement

The concept of SoPC based on COTS FPGA technology may provide several benefits. When using commercial FPGAs, designers can concentrate on the development of the functionalities the system has to provide in response to user requirements. Silicon design and manufacturing is demanded entirely to FPGA vendors that are making devices available even before the designs the silicon will implement are ready. In case of low volume productions, significant cost savings are possible as well as shorter time-to-market with respect to ASICs.

Modern COTS FPGAs are based on programmable, rewritable memories (either using SRAM or Flash cells) for storing the configuration describing the design the device implements. The capability of reprogramming the FPGA device when already in the field offers to designers the possibility of servicing the system without replacing any hardware components. Furthermore, new design paradigms like configurable computing can be employed, thus making possible the implementation of innovative designs. An interesting example of these designs can self-modify to react to a changing environment [5].

When considering COTS FPGAs for space- or avionic-based applications, designers must take into account that ionizing radiations (e.g., neutrons and heavy ions) permeate the environments where the electronic systems operate, possibly provoking Single Event Effects (SEEs) [6][7]. The impact of SEEs in the design the FPGA implements should be evaluated and the needed countermeasure should be employed to avoid failures [1][2]. Radiation-hardened FPGAs are not considered in this work, as they typically do not provide resource and performance levels comparable to those of devices intended for mainstream applications, and their cost is often outside the affordable budget for many applications.

When devising SEE-mitigation techniques for COTS FPGAs, researchers exploited hardware redundancy [1], and/or time redundancy [2]: the modules in the design are replicated and SEE-filtering modules (e.g., voters or guardian gates [2]) are added to the designs. When SoPCs are considered, these approaches entail the systematic modification of all the modules in the system: the processor cores, the memories and the ad-hoc hardware accelerators. As a result, significant hardware overhead has to be taken into account. As an example, when SRAM-based FPGAs are considered, designers have to account for at least a 3x hardware overhead as all the modules of the design (both the combinational and the sequential logic) have to be replicated three times and voters have to be added to account for SEEs affecting the device configuration memory and the user memory [1].

By considering the nature of the different components inside a SoPC it is possible to devise new, less expensive solutions. In particular, by recognizing the peculiarity of the processor cores embedded in a SoPC it is possible to exploit redundancy techniques alternative to brute force hardware replication, which allow reducing the hardware overhead.

In this paper we propose a design flow to mitigate the SEEs possibly affecting processor cores embedded in SoPCs that takes into account the specific characteristics of these devices. The design flow provides an automatic way for applying SEE-mitigation to any new designs. From the hardware point of view, processor cores are automatically duplicated and suitably connected, and a small piece of hardware is added. From the software point of view, the original code the processor core runs is supplemented with two custom-developed interrupt service procedures. The main benefits of the proposed flow are a reduction of the required hardware resources, and the possibility of easily and quickly developing dependable processor cores for inclusions in new SoPCs. Furthermore, the design flow is general: it exploits common features available in any recent processors (e.g., an interrupt controller) and it is structured in such a way that the custom-design hardware modules it requires are reusable.

3.Assumptions

For the sake of this paper we assume that an unhardened SRAM-based FPGA is used for implementing the SoPC. The faults of concern for this type of device are:

1. SEEs in the device configuration memory that correspond to the persistent modification of bits of this memory. The faults, known as *Single Event Upsets* (SEUs), consist in the modification of the functionality of the processor core the FPGA implements as they modify the way in which the FPGA resources are configured. These faults have persistent effects, since when hitting the configuration memory, they remain latched until the device is reconfigured.
2. SEUs in the device user memory implementing the processor registers and caches. This type of faults is transients as it persists in the processor until the affected memory element is rewritten.

In case Flash-based devices are used, the configuration memory can be considered as immune against SEE, and therefore only SEUs in the user memory have to be considered. No particular restrictions are required by our method for the processor the SoPC uses. The only requirement is that the processor cache system exploits a write-through policy for updating the content of the data cache.

4. Proposed solution

4.1.Hardening architecture

The hardening solution we follow is mainly based on a combination of lockstep for error detection, checkpointing and rollback for error recovery. Following this approach, and exploiting the specific characteristics of a SoPCs, hardening can be done in the following steps:

1. The processor in the system is duplicated, and the two replicas are connected in a lockstep fashion [3][4], as depicted in figure 1, so that the new system is composed of two exact copies of the processor: the *master* and the *checker*.
2. A monitor module (*checker logic*) is added to the system, that:
   a. Detects the occurrence of SEU-induced misbehaviors either in the master or in the checker.
   b. In case no errors are detected, it periodically issues an interrupt request to be handled by the master processor, during which it saves a copy of the system's context (the content of the processor's registers and its memory).
   c. In case an error is detected, it issues an interrupt request to restore the system's context to the previously save one, so that the two processors are rolled back to a previously good state.
3. The FPGA floorplan is divided in three non-overlapping areas: one for placing the master CPU, one for the checker CPU and one for the checker logic.
4. The master CPU, the checker CPU and the checker logic are placed in the dedicated areas through suitable placement constraints for the place & route tools.
5. The software is properly modified by adding two interrupt service procedures to support the saving and restoring of the system context when required by the checker logic.

In the following, the various aspects of the proposed solution are detailed.

4.2.Fault detection and correction

SEUs may affect the configuration and user memories of the checker logic, and that of the two CPU replicas, as well as the interconnection between them.

In our approach, as the areas where the above modules are placed are not-overlapping, SEUs may alter only one of the tree modules at a time. As a result, under the single fault assumption we may consider that only one faulty module is present in the system at any given time.

To mask the propagation of faults in the checker logic, and the interconnections between the checker logic and the CPU replicas, we locally exploit the Triple Modular Redundancy (TMR) approach. As the checker logic area is negligible with respect to that of the CPU, the total overhead of using such approach with respect to the size of the whole system is negligible.

Faults in the two CPU replicas are handled by the proposed mitigation scheme, which exploits lockstep execution for error detection and checkpoint with rollback recovery for correction of SEUs in the user memory [3][4]. On-demand configuration memory scrubbing is used for correcting persistent faults in the configuration memory.

4.2.1.Lockstep, checkpoint and rollback

The lockstep technique is an error detection mechanism that adopts two identical processors (a *master* and a *checker*) with different roles. The master and the checker execute the same program but only the master interfaces with the other system hardware components (such as memory, peripherals, and other devices). The outputs of both CPUs are connected to a *checker logic* that compares the results and possibly rises up a mismatch signal. The name lockstep is due to the fact that both processors have to lock their execution at the same time and then they restart together in synchronous way; in this way the two processors' state are always the same for each clock period in which the application is executed.

Lockstep is particularly interesting as it allows implementing a safe system with minimum hardware overhead (the checker logic); moreover, it does not require any modification to the software the processor runs. Therefore, designers can concentrate on the functionalities the system has to provide, while neglecting dependability issues. Dependability issues will then be tackled transparently to the functionalities the system implements.

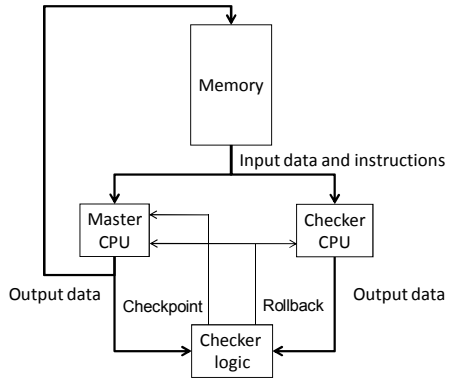

Figure 1. Typical architecture of a lockstep system enriched with checkpoint and rollback capabilities

Figure 1 depicts the typical architecture of a system exploiting lockstep, enriched with the additional features we developed for surviving the occurrence of transient errors.

The data/instruction memory is connected to both processors so that both processors always receive the

same input. The master CPU is the only one allowed modifying the memory by sending the output data it computes. Both processors send their output data to the checker logic. The checker logic compares the results and signals possible mismatches according to the following scheme:

1. In case of no mismatch, if a certain activation condition is met (e.g., at each write operation, or every a given number of seconds), the checkpoint signal is issued. The signal triggers an interrupt routine that saves in the memory the content of all the registers of the master CPU, as well as the data segment of the application through a memory-to-memory DMA transfer. The stored information are called processor's *context*.
2. In case of mismatch, the rollback signal is issued. The signal triggers an interrupt handling routine whose purpose is to restore the previously saved context on both processors. Upon rollback execution, the two processors are step back to a previously-saved fault-free state, and therefore the part of program execution corrupted by the detected SEU is discarded and the execution is resumed from the last known good state.

4.2.2.Architecture of the checker logic

The checker logic is a custom-designed module in charge of monitoring the operations performed by the two processors of the lockstep architecture of figure 1. The functions it implements are the following:

1. It decodes every memory access cycle (both write and read cycles), and latches the values on the data, control and address buses of both processors, being B_{Master} and $B_{Checker}$ the latched values.
2. In case B_{Master} differs from $B_{Checker}$ the rollback signal is asserted, and it is de-asserted only when the master processor acknowledges the interrupt request.
3. In case B_{Master} is equal to $B_{Checker}$ and an activation condition is met the checkpoint signal is asserted, and it is de-asserted only when the master processor acknowledges the interrupt request.
4. In case two successive rollback requests are issued before a new checkpoint is reached, the control logic triggers the reconfigure signal to have the FPGA device reprogrammed. As detailed later, this operation is needed for recovering from persistent errors.

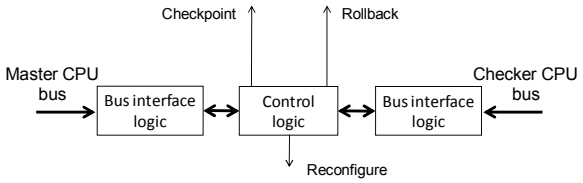

Figure 2. Architecture of the checker logic

The architecture of the checker logic is depicted in Figure 2.

The *bus interface logic* decodes the bus cycle the processor executes, latches the value of the address, data and control bus, and forwards these values to the control logic. The *control logic* implements the consistency checks, the checkpoint activation policy, and the rollback activation policy.

The checker logic is highly reusable. The control logic is transparent to the processors used in the lockstep architecture, and therefore can be used in any design. The only effort requested to designers is to adapt the bus interface logic to behave according to the bus specification of the processor of choice. Moreover, being the design modular, it can easily fit also in architectures where two different processors are connected in lockstep fashion, thus targeting any common mode failures in the processors.

4.2.3.Checkpoint execution

The operation of saving the system's context is known as *checkpoint* operation.

In this work we adopt the strategy of executing checkpoint operations periodically. A new checkpoint is taken upon the completion of a write operation and after Δ clock cycles since the last checkpoint. In this way, we reduce the performance overhead due to checkpoint execution in the case of applications that execute a higher number of write instructions.

The checkpoint routine is an interrupt service procedure that has to be added to the processor code segment, which has to:

1. Save the CPU registers to a memory devoted to context storage.
2. Make a copy of the data memory using a memory-to-memory DMA transfer.

A critical aspect of the checkpoint execution is the time spent for saving all the information needed. As far as processor's registers are considered, in case the processor exploits register windows, this task can be easily performed by increasing the current window pointer. Conversely, register-to-memory copy instructions have to be executed.

When applications with large memory footprints are considered, although DMA transfer is the most convenient approach to perform context saving, the time spent for completing the memory transfer may not be negligible. Therefore, the checkpoint mechanism must be tuned to find the proper balance between the time overhead due to context saving and the amount of re-computation performed in case of error detection.

4.2.4.Restoring the system's context

When activated, the rollback mechanism sends back the entire system to a previously consistent state. The basic rollback routine is an interrupt service procedure

that has to be added to the processor code segment, which has to:

1. Replace the main memory with the safe copy, through DMA transfer.
2. Load the previously saved copy of the CPU registers from the memory area devoted to context storage.
3. Invalidate all caches.

4.2.5.Correcting persistent faults

As suggested in [1], periodically rewriting the configuration memory using the operation known as memory *scrubbing* is a cost effective approach to correct persistent SEUs affecting the FPGA configuration memory, and any SoPC based on SRAM FPGAs aiming at working in a radioactive environment is likely to include a memory scrubber. In our flow we assume the presence of the memory scrubber, and in case the checker logic detects the execution of two consecutive rollback operations before a new checkpoint is triggered, it assumes that the fault is persistent, and thus it triggers the memory scrubber. Upon scrubbing completion, the processors are reset and the whole computation re-executed.

The rationale behind this approach is that in case the SEU is not corrected through rollback, it may indicate that the fault is persistent and therefore configuration memory scrubbing is needed to correct it.

4.3.Automated design environment

A major issue for designers of hardened systems is the minimization of design costs, including those for validating the correctness of the design and of the implemented hardening mechanisms. For this reason, designers require not only suitable design techniques from the point of view of hardening capabilities and hardware costs, but are particularly interested in solutions, whose implementation can be effectively automated.

Following this idea, we propose an automated design flow, which can effectively support the designer of safety-critical SoPC-based systems. The flow, written in C language, and accounting for about 2,000 lines of code, automates the modification of the HDL design to reflect the proposed hardening solution: moreover, it enriches the system software by inserting the interrupt routine managing checkpoint and rollback operations.

5.Experimental results

This section describes the experiments we performed to assess to assess the effectiveness of the proposed design flow. We took as a case study a SoPC based on the Leon 2 processor [8], which is the common denominator of recently developed payload for several space missions, as for example the Venus Monitoring Camera [9].

In the experiments we analyzed two parameters: the area overhead our method entails with respect to the state-of-the-art approach based on pure TMR, and the fault detection and correction capabilities of the architecture resulting from our design flow.

5.1.Overhead analysis

To evaluate the overhead of the proposed approach, and to compare it with the alternative state-of-the-art solutions we first synthesized the Leon 2 processor core, we then obtained its TMR version using the XTMR tool [10], and we finally compared its resource occupation (in terms of logic cells and memory blocks) with that of the architecture obtained according to the proposed design flow. We adopted a Xilinx Virtex device for synthesis, and place and route purposes. The obtained results are reported in Table 1.

	Logic cells [#]	Memory blocks [#]
Leon 2	4,019	25
Leon 2 TMR	14,067	75
Proposed approach	8,473	50

Table 1. Comparisons among different synthesis results

As the reader can observe, the TMR version requires more than three times the resources occupation of the original Leon 2 version, thus potentially imposing a significant limitation on the possibility of integrating all the system in the same device. On the other hand, a lower overhead is needed when the architecture obtained from the approach proposed in this paper is considered: this overhead accounts for the duplication of the processor as required by the lockstep, plus the introduction of the logic for managing error detection and for issuing checkpoint and rollback requests that, as explained before, is mitigated against SEEs using TMR. The most relevant benefit of the approach is probably related to the saving in terms of number of memory blocks, which are typically available in much lower quantity than logic cells.

In order to assess how the proposed design flow behaves under different usage scenarios, we considered four different software applications, whose characteristics are reported in Table 2. In the same Table the reader can find for every application the execution time, the size of the code and data segments, as well as the size of the two interrupt procedures that implement checkpoint and rollback. As the reader can observe, the impact of the two procedures is negligible on the memory footprint of the system. In our experiments we choose to perform checkpoint every 300 write cycles, obtaining the overhead figures reported in Table 2. As the reader can observe, the time overhead ranges from 17% to about 54%, and mainly depends on the amount of data that must be saved during DMA transfer. The Leon processor offers powerful features to implement checkpoint and rollback in efficient ways. In particular, register saving is implemented using the register window mechanism and the instructions the

Leon provides for saving and restoring the content of register windows.

Program	Clock cycles [#]	Checkpoint overhead [%]	Code Size [bytes]	Data Size [bytes]
Ellipf	9,885	27.5	860	284
FIR	6,301	17.7	992	136
Kalman	32,840	53.8	1,348	2,480
Matrix	10,768	18.3	496	300
Checkpoint procedure	-	-	108	-
Rollback procedure	-	-	136	-

Table 2. Characteristics of the adopted benchmarks

5.2.Dependability analysis

To assess the impact of soft errors in the user memory, we injected 10,000 randomly selected SEUs in the processor registers, and in particular the pipeline registers. We decided to attack these registers as they represent one of the most critical parts of the processor, and therefore these experiments allow understanding the capabilities of our approach in a worst-case situation. The emulation-based fault injection tool presented in [11] was used for this purpose. From the experiments, we observed that on the average:

1. 84% of the injected faults are either effect-less (i.e., they disappear few clock cycles after their injection without visibly affecting the system) or detected and corrected properly.
2. 15% of the injected faults triggers an error detection mechanism embedded in the processor (i.e., the illegal instruction trap). To handle correctly this case, we had to extend the checker logic to make it able to recognize the execution of an unexpected trap, and to force the execution of a rollback operation.
3. 1% of the injected faults triggered the on-demand configuration memory scrubbing as they were classified by the checker logic as persistent errors. We investigated this situation that we originally expected only in the case of SEUs in the configuration memory, and we found that some SEUs may remain latent in the processor and manifests themselves only after the checkpoint is executed. As a result, the rollback is performed using a faulty context, forcing the processor to fail again as in the case of persistent faults in the configuration memory.

In order to evaluate the impact of faults in the configuration memory, we analyzed the design using the STAR tool [12] and we did not identify any common-mode fault that may overcome the adopted SEU-mitigation schemes.

6.Conclusions

In this paper we presented a possible solution to mitigate processor cores embedded in SoPCs implemented using COTS FPGAs. The approach is based on a suitable mix of known techniques, and has been customized to the specific characteristic of programmable devices currently on the market. Moreover, the design flow can fully be automated, thus freeing the designer from the burden of implementing and validating the required hardening mechanisms. We evaluated our approach on a case study based on the Leon processor, and we observed that resource overhead is significantly reduced with respect to state-of-the-art TMR-based techniques, while still maintaining a high error detection and correction capability.

References

[1] C.C. Yui, G.M. Swift, C. Carmichael, R. Koga, J.S. George, “SEU mitigation testing of Xilinx Virtex II FPGAs”, IEEE Radiation Effects Data Workshop, pp. 92-97, 2003

[2] S. Rezgui, J.J. Wang, E.C. Tung, B. Cronquist, J. McCollum, “New Methodologies for SET Characterization and Mitigation in Flash-Based FPGAs”, IEEE Trans. Nuclear Science, Vol. 54, No. 6, 2007, pp. 2512-2524

[3] N. S. Bowen, D. K. Pradhan, “Processor and memory based checkpoint and rollback recovery”. IEEE computer, 1993, 26(2):22-29.

[4] J. R. Walker, C. Papachristou, “A Rollback Recovery System for Embedded FPGA Processors", MAPLD International Conference, 2006.

[5] “Self-repairing and tuning reconfigurable electronics: real world applications”, Tutorial at the NASA/ESA Conference on Adaptive Hardware and Systems, 2008

[6] G.R. Allen, G.M. Swift, “Single Event Effects Test Results for Advanced Field Programmable Gate Arrays”, IEEE Radiation Effects Data Workshop, pp. 115-120, 2006

[7] J. George, R. Koga, G. M. Swift, G. Allen, C. Carmichael, C.W. Tseng, “Single Event Upsets in Xilinx Virtex-4 FPGA Devices”, IEEE Radiation Effects Data Workshop, pp. 109-114, 2006

[8] www.gaisler.com

[9] http://www.mps.mpg.de/projects/venus-express/vmc/

[10] www.xilinx.com

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

[12] L. Sterpone, M. Violante, R. Harboe Sorensen, D. Merodio, F. Sturesson, R. Weigand, S. Mattsson, "Experimental Validation of a Tool for Predicting the Effects of Soft Errors in SRAM-based FPGAs", IEEE Transactions on Nuclear Science, Vol. 54, No. 6, Part 1, December 2007, pp. 2576-2583