

Combined software and hardware techniques for the design of reliable IP processors

M. Rebaudengo, L. Sterpone, M. Violante
Dip. di Automatica e Informatica
Politecnico di Torino
{reba, sterpone, violante}@polito.it

C. Bolchini, A. Miele, D. Sciuto
Dip. di Elettronica e Informazione
Politecnico di Milano
{bolchini, miele, sciuto}@elet.polimi.it

Abstract

In the recent years both software and hardware techniques have been adopted to carry out reliable designs, aimed at autonomously detecting the occurrence of faults, to allow discarding erroneous data and possibly performing the recovery of the system. The aim of this paper is the introduction of a combined use of software and hardware approaches to achieve a complete fault coverage in generic IP processors, with respect to SEU faults. Software techniques are preferably adopted to reduce the necessity and costs of modifying the processor architecture; since a complete fault coverage cannot be achieved, partial hardware redundancy techniques are then introduced to deal with the remaining, not covered, faults. The paper presents the methodological approach adopted to achieve the complete fault coverage, the proposed resulting architecture, and the experimental results gathered from the fault injection analysis campaign.

1 Introduction

In the most recent years, given the relevant computational power of IP cores, the design of reliable systems by means of software techniques received a lot of attention, due to the interesting possibility to use a commercial processor core, without requiring any customization. Indeed, such an advantage is very important when considering time to market, the flexibility of the software, as well as the independence of the system from a specific IP core vendor. Nevertheless, the drawback of a pure software approach is the impossibility of achieving a complete fault coverage [4], due to the presence of a limited, but not empty, set of faults that may cause the processor not to behave correctly, without signaling such a critical situation. As a consequence, in safety-critical systems, where it is mandatory to be able to guarantee and assess a complete fault coverage, a combined software-hardware approach needs to be adopted. Our approach aims at introducing a hardware fault detection mechanism for those errors that may not be detected by means of software techniques; costs and benefits will be evaluated, achieving the desired complete fault coverage.

It is worth noting that this approach is not only the result of the systematic analysis of software techniques for hardware fault detection and tolerance, but it is also a mixed hardware/software co-design solution for the design of reliable systems. In fact, the information on fault coverage derived from the qualitative, methodological approach allows an estimation of costs and benefits of the pure software techniques, whereas overheads and fault coverage for classical hardware techniques are well-known. Therefore, in a reliability-aware hardware/software co-design flow [3] where a complete fault coverage is required and

flexibility and area constraints drive the solution space exploration, the achieved architectural result is a valid solution. We can consider the 100% hardware redundancy technique as the solution at one end of the solution space providing a complete fault coverage. On the other end of the solution space, with a complete fault coverage, there is the application of all available software techniques, supported also by a partial hardware redundancy to cover all faults, identified as *all-software and partial hardware* approach. Other solutions lie in between, with a different mix of software and hardware techniques and still providing a complete fault coverage, characterized by various overheads in terms of area, code-size, performance degradation, \dots ; in this paper we analyze the all-software and partial hardware approach achieving a complete fault coverage and maintaining area overhead at a minimum, whereas in the future we will investigate the other possible intermediate solutions.

The main contribution of this paper is a revised version of the preliminary qualitative analysis of soft errors presented in [4], and a modified application of redundancy techniques with respect to those presented in [2], driven from the results of the presented analysis. As experimental results will show, the methodological approach enabled a more specific hardware customization, overcoming the limitation of the previous work. An architecture-independent methodology is presented, whereas the experimental phase refers to the Leon2 processor architecture [9].

The rest of the paper is organized as follows. Section 2 briefly introduces the adopted fault model and presents the proposed combined approach designed to cover all possible errors caused by SEUs, discussing the software and the hardware techniques used to pursue the desired goal. The resulting solution has been validated from a theoretical point of view, nevertheless a fault injection campaign has been also carried out, to further verify that all faults are covered and new ones are not introduced; the results are reported in Section 3. Final remarks and future work close the presentation.

2 The proposed approach

The approach here presented, starts from conclusions drawn in [2] and proposes an enhancement based on a more systematic analysis of the fault/error relationship. The methodology combines software and hardware techniques for fault detection to introduce reliability properties into software modules of critical embedded systems. The methodology is mainly based on the software approach due to its benefits – acceptable error coverage and reduced design costs and overhead, coupled with a performance degradation and code size growth – and exploits a partial hardware replication to provide the detection of faults that are not covered by the software techniques. The adopted fault model is introduced in the following, then an evaluation of the software techniques will be presented. The goal is the identification of the areas of the processor whose functionalities are not completely covered, leading to the introduction of partial hardware replication.

2.1 Fault model

Single programmable logic device, such as System-on-Chip (SoC) and Field Programmable Gate Arrays (FPGAs), are today commonly used to implement embedded systems. Such devices are susceptible to errors resulting from the perturbation of memory cells, provoked by ionizing radiation. An example of this effect is the Single Event Upset (SEU), i.e., an

undesired change of state in a bistable or storage element, also referred to as a *soft error*. SEUs are a major cause of concern in a space environment, but they have also been observed at ground level [7]. In a combinational circuit, e.g., an arithmetic logic unit (ALU), ionizing radiation may induce a transient voltage pulse that can lead to incorrect output. The modification of the memory cell is not permanent, since a successive write operation can override the erroneous content and replace it with a new one. The proposed approach aims at providing a complete fault coverage for any single SEU effect in the storage elements of a processor, in a SoC or in a FPGA. For the analysis of the software techniques, the considered fault is modeled at an higher abstraction level, as further detailed in the next subsection.

2.2 Software techniques

Software techniques for fault detection are a well explored issue in literature ([10, 5, 1, 8]). The basic idea is to apply temporal and information redundancy strategies on the program code, modified by introducing additional instructions and variables to perform equivalent computations more than once and to compare results. Three main techniques are considered:

Data processing instruction replication ([10, 5]). Replication of all the variables of the program and all the data processing instructions. This replication aims at performing redundant data elaborations to check redundant results.

Self checking block signature ([1, 8]). Static control of the execution flow; the technique assigns to each basic block of the program (the maximum branch-free block of instructions) a static signature and introduces, at the beginning and at the end of each basic block, two instructions for checking and updating the runtime signature stored in a global variable against the static signatures.

Conditional branch instruction replication ([10]). Replication of the branch instruction in each branch direction, aiming at performing a run-time check of the direction taken during the execution of a branch instruction.

This set of techniques is, though, not sufficient to cover all possible errors caused by SEU faults, as it will be shown in the following. The first attempt to classify such open issues has been adopted as the starting point ([4]), to carry out an architecture-independent analysis, able to fit different processors. For this purpose, the adopted architecture is described in a behavioral fashion, to maintain a high level of abstraction and to describe the essential aspects of any processor without considering structural aspects that are peculiar to the implementation of one processor. The model is composed of a *program executor* and two memories (a *data memory* and an *instruction memory*). The program executor is a high level description of a single processor architecture, modeled only with its workflow, shown in Figure 1, composed of several basic operations that are executed sequentially step-by-step, for each instruction of the program code. While considering the imperative programming paradigm, the instructions that the model executes can be classified into two classes: *data processing instructions* and *control instructions*. The former are executed sequentially starting from the entry point of the program, while the latter, only, can modify the instruction flow by performing a jump to any other instruction.

When considering the adopted architecture, the fault may be modeled as a corruption of the execution of a single step of the workflow or as a corruption of the content of one

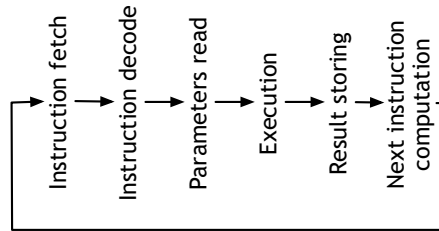


Figure 1. Program executor workflow

of the two memories. When considering the *program executor*, the result of the physical fault at the behavioral abstraction level is represented as a corruption of a value, and the effect is an erroneous execution of a step of the processor workflow. If the fault corrupts the memories, an error is generated and is detectable when the corrupted value is used for a computation. When an error affects the program execution, the data or the control flow of the program is corrupted. When a value used for the data elaborations is corrupted, an error is generated in the data flow. When an error affects the control flow, it causes an erroneous evolution of the execution flow; in particular, the effect is a wrong jump performed within the instruction code and, thus, the execution continues with a wrong instruction. More in detail, errors affecting the control flow can be classified as follows:

- during the execution of an instruction inside a basic block, instead of the current instruction, a jump may be performed to another instruction either (a) in the same basic block, or (b) in a different basic block.
- during the execution of the control instruction at the end of a basic block, a jump may be performed either (a) to the beginning of a wrong basic block, or (b) to the middle of a basic block.

Given the results of the analysis of the possible faults in the architecture and of the corresponding errors affecting the program execution, it is possible to study the cause/effect relationship between faults in the architecture and errors in the program execution. When considering the adopted program executor, it is possible to state that both the functionality corrupted by the fault and the type of instruction that the processor is executing are responsible of the type of generated error and its effects on the program execution. The fault/error relationship is shown in Figures 2 and 3, where a specialization of the above listed errors is used: when executing a data processing instruction, a fault generally causes an error in the data flow; in case the computation of the next instruction is corrupted or the current instruction is transformed into a control instruction, an error may be generated also in the control flow. On the other hand, when executing a control instruction, the effect of the fault is generally an erroneous evolution of the control flow.

When the fault affects one of the two memory elements the error will manifest its effects when the corrupted value is used by the executor. If the data memory is affected, the error will corrupt the data flow if the value is used during the execution of a data processing instruction; otherwise the control flow is affected. When the instruction memory is corrupted, an instruction is transformed into a new one: therefore, the error will affect

Corrupted phase	Generated errors
<ul style="list-style-type: none"> ○ Instruction fetch ○ Instruction decode 	<ul style="list-style-type: none"> ○ A data processing instruction is transformed into: <ul style="list-style-type: none"> □ another data processing instruction: a wrong result is produced □ a control instruction: an erroneous jump to a random target is performed <ul style="list-style-type: none"> ▷ in the same basic block, or ▷ in a different basic block
<ul style="list-style-type: none"> ○ Parameters read ○ Execution ○ Result storing 	<ul style="list-style-type: none"> ○ Instruction execution leads to a wrong result that is stored in memory
<ul style="list-style-type: none"> ○ Next instruction computation 	<ul style="list-style-type: none"> ○ An erroneous jump to a random instruction occurs: <ul style="list-style-type: none"> □ in the same basic block, or □ in a different basic block

Figure 2. Errors possibly caused by faults affecting a data processing instruction

Corrupted phase	Generated errors
<ul style="list-style-type: none"> ○ Instruction fetch ○ Instruction decode 	<ul style="list-style-type: none"> ○ The control instruction is transformed into: <ul style="list-style-type: none"> □ a data processing instruction: a wrong result is produced and no jump is performed □ another control instruction: an erroneous jump to a random target is performed <ul style="list-style-type: none"> ▷ in the same basic block, or ▷ in a different basic block
<ul style="list-style-type: none"> ○ Parameters read ○ Execution ○ Next instruction computation 	<ul style="list-style-type: none"> ○ Jump target computation error: an erroneous jump to a random instruction occurs: <ul style="list-style-type: none"> □ in the same basic block, or □ in a different basic block ○ Branch direction evaluation error: wrong branch direction is taken ○ The execution of a jump instruction is affected: no jump is performed
<ul style="list-style-type: none"> ○ Result storing 	<ul style="list-style-type: none"> ○ No instruction is performed: the fault causes no error because it is not activated

Figure 3. Errors possibly caused by faults affecting a control instruction

the data flow if a data processing instruction is transformed into another data processing instruction; otherwise also the control flow is affected.

The presented fault/error relationship provides the starting point to perform a critical analysis of the most interesting software techniques for fault detection. The purpose is to identify the errors each technique covers and the errors not covered by any technique, to determine the functionalities of the processor not completely dealt with by the considered software techniques.

The qualitative analysis shows that by means of the adopted techniques a good capability of error detection is achieved for the data flow of the program. In fact, due to the replication of all the data processing instructions and of all the variables, the fault affecting the processor will corrupt a value and the checking instruction will detect an incongruence between two redundant values. On the other hand, the study shows that the techniques present some limitations in the detection of errors that affect the control flow. Even if the techniques of block signature and conditional branch instruction replication make it possible to identify erroneous jumps among different basic blocks and wrong evaluations of branch directions, a partial error detection capability is obtained in case of erroneous jumps to instructions of the same basic block. In fact the block signature technique is not able to detect this kind of error; moreover, the replication of the data processing instructions makes it possible to detect this kind of error only if the erroneous jump generates an incongruence in the data flow, i.e., if only one of the two replicas of an instruction is executed.

Another situation that cannot be covered by software techniques is related to the control transfer performed when an exception is thrown due to a trap or an interrupt. In this scenario the block signature technique cannot be applied because the jump is not performed by an explicit instruction but is requested by an interrupt or a trap. It is worth noting that this scenario is not taken into account for this study because no specific operating system is considered, and the management of these issues strictly depends on it.

The results of the analysis show that the functionalities not covered by the software techniques are those that cause an erroneous jump to an instruction of the same basic block. The fault that may originate this error is in the next instruction address computation. Moreover, the decoding phase may be involved in scenarios that lead to the same kind of error: in case an instruction is transformed by a fault into a control instruction, a random jump may be performed to another instruction of the same basic block. Finally, this kind of error may be generated by faults affecting the checking instruction introduced by the replication of the data processing instructions; in fact, when this technique is applied, a checking instruction is introduced into the code and it is not possible to control its execution by means of the block signature. Thus, it is possible to state that a part of the processor functionalities is not completely covered by means of the software approach. In the next subsection it will be shown how to overcome the highlighted limitations.

2.3 Hardware techniques

By referring to an architecture model, the adopted program executor consists of several modules that implements the described basic functionalities. Therefore, it consists of a control module composed by units such as an instruction fetch unit or an instruction decode units, and a data-path composed by units such as an ALU or memory access interfaces. By considering the architecture model, it is possible to state that the units whose functionalities are not completely covered by the software techniques; such units are the program counter management unit, the decoding unit and the branch management unit.

When considering the features of the adopted fault model – the soft error usually affects storage elements – it is necessary to act on the registers of the uncovered units by means of hardware replication and comparison. Therefore, the subset of registers to be replicated depends on the structure of the units that implement the selected functionalities. It includes the program counter and the instruction register; according to the processor structure, other control registers should be duplicated, for instance the registers used for transmitting the jump address between the several units. Moreover, when considering a pipelined processor, several pipeline stage registers containing information related to the instruction word, the program counter and the jump address should be replicated.

The partial hardware replication here applied consists in duplicating the identified subset of registers; for each one of them, a redundant register is added. The basic register is used for the nominal computation and the redundant one is used for store a copy of the value. A checker is used to compare the contents of the two replicas: when inconsistencies are observed, the system can be held and reset to prevent wrong values from being propagated. It is worth noting that the adoption of this strategy of hardware replication does not affect processor performance and guarantees concurrent error detection without additional computational overhead, although it implies an overhead in terms of area.

3 Methodology evaluation

In order to evaluate the proposed hardening methodology we analyzed a case study based on a microprocessor system considering a fault injection system based on an emulation environment implemented on a SRAM-based FPGA board [6]. This environment is able to inject SEUs within all the memory elements of a microprocessor core. The evaluation flow we adopted consists first in hardening the microprocessor system considering the proposed Program Executor workflow model. Then, several fault injection campaigns, aimed at demonstrating the effectiveness of the proposed hardening methodology, are performed.

3.1 Case study

The case study we adopted is the LEON-II microprocessor core [9], a processor based on an integer unit (IU) that implements SPARC integer instructions. It has several features such as a 5-stage instruction pipeline, separate instruction and data cache interface and an architectural support for 2 to 32 register windows. The proposed program executor workflow is adaptable to the considered architecture, where the program executor's phases model the pipeline stages and the program executor's modules implementing the basic functionalities model the registers and the functionality units. In details, the instruction fetch stage corresponds to the pipeline fetch, the instruction decode and the parameters read model the decode stage, the execution stage model the pipeline execute, finally the result storing and the next instruction computation stages model the memory and the write states, respectively.

We applied the analysis methodology described in Section 2.2 to the LEON-2 processor, and we identified those components that, in the considered processor, implement the program counter management unit, the decoding unit and the branch management unit. In order to detect those occurrence of SEUs in these units not covered by software rules, the following registers must be hardened: the branch address register, the jump address register, the program counter, the instruction registers, the trap control register, the result register, the annul register (used to flush the pipeline in case of hazard), and the register controlling the execution of instructions that span over more than one clock cycle. For the sake of this work, we adopted a simple duplicate-and-compare approach: we duplicated the registers, and we inserted a checker that signals the occurrence of any SEUs affecting them. The adopted solution does not affect the performance of the processor, while it introduces a limited area overhead, which can be estimated to be about 2%.

3.2 Experimental results

To assess the effectiveness of the proposed approach we exploited the environment developed in [6] that provides the required accuracy since it allows accessing all the memory elements the processor embeds with a suitable time resolution. The environment consists of a FPGA board used to emulate an instrumented model of the considered processor, and of a Personal Computer that hosts the FPGA board and runs the software managing the fault injection experiments. The model of the processor is enriched (i.e., instrumented) with suitable hardware components allowing observing and controlling each memory element, through which SEUs are injected, and SEU effects are observed. The fault injection experiments have been performed considering 3 benchmark programs:

1. Finite Impulse Response filter (FIR): A filtering of 16 fixed-point samples.
2. Fifth Order Elliptical Wave Filter (ELPF): An elliptic filter over a set of 32 fixed-point samples.
3. Kalman filter (KLMN): A Kalman filter on a set of 16 fixed-point samples.

For the considered benchmark applications, we compared 3 implementations strategies:

- a. Plain: It is the plain version of the considered application. Hardware and software hardening techniques are not applied.
- b. All-software: It is the software hardened version of the considered application. This is obtained using only software techniques as it is explained in Section 2.2.
- c. All-software and partial hardware: It is the hardened version of the application, obtained using the combination of software and hardware techniques previously described.

We analyzed the dependability of the three implementations by performing the injection of 1,000 randomly selected faults in each of the 5 pipeline stages that compose the considered processor: the fetch stage (FE), the decode stage (DE), the execution stage (EX), the memory-access stage (ME), and the write-back stage (WE). The attained results are reported in Figure 4, where the number of SEUs provoking the program to compute wrong results are listed, along with the execution time, the data size, and the code size. As the reader can observe, about 10% of the injected faults lead the plain version of the programs to produce wrong results. The considered faults are thus relevant, and effective countermeasures are needed. The programs hardened according to the all-software approach show promising results in terms of fault detection, we indeed observed a reduction of two orders of magnitude in the number of wrong results. However, few faults still escape the software detection mechanisms, confirming the necessity to introduce a further hardening mechanism. The results gathered when considering the combined all-software and partial hardware approach show that the number of faults causing wrong results is reduced to zero, confirming the analysis described in Section 2. These experiments outline the effectiveness of integrating techniques based on software modifications, with techniques that require limited hardware modifications. However, the achieved improvement of system's dependability comes at a cost of increased execution time (due to the time redundancy and the additional instruction for checking purposes we inserted in the software), memory overhead (due to the replicated data and instructions), and area overhead (due to the replicated registers). For the considered benchmarks, the average increase of the execution time is about 6 times, the memory overhead is about 4 times, while the area overhead is about 2%.

program	version	pipeline stage					total	exec time	data size	code size	exec time ratio	data size ratio	code size ratio	memory ratio
		FE	DE	EX	ME	WR								
FIR	plain	190	142	67	39	11	449	6235	200	860	1.00	1.00	1.00	1.00
	sw	3	0	0	0	0	3	29293	400	2864	4.70	2.00	3.33	3.08
	sw+hw	0	0	0	0	0	0	29293	400	2864	4.70	2.00	3.33	3.08
ELPF	plain	210	110	69	36	41	466	9854	164	724	1.00	1.00	1.00	1.00
	sw	3	0	0	0	0	3	110955	328	4408	11.26	2.00	6.09	5.33
	sw+hw	0	0	0	0	0	0	110955	328	4408	11.26	2.00	6.09	5.33
KLMN	plain	220	157	99	64	16	556	32811	6472	1216	1.00	1.00	1.00	1.00
	sw	2	0	0	0	0	2	258445	12944	6184	7.88	2.00	5.09	2.49
	sw+hw	0	0	0	0	0	0	258445	12944	6184	7.88	2.00	5.09	2.49

Figure 4. Experimental results: for each benchmark and implementation strategy we report the number of undetected faults. Costs are also reported in terms of time and memory (times are in clock cycles and code size is in byte).

4 Concluding remarks

The proposed solution is aimed at covering with hardware techniques the faults not addressed by means of software techniques for fault detection. Previous approaches have been refined, from the methodological point of view, thus allowing an efficient application of software and hardware techniques for fault detection, in terms of overheads and benefits. The result is a combined set of techniques, that maintains the flexibility of the software, as well as the possibility to introduce such redundancy only for critical tasks, and introduces the minimal amount of hardware redundancy and processor modification, needed to provide a complete fault coverage, when desired. The proposed solution, derived from the qualitative and systematic analysis of fault/error relations, carried out from an architectural independent point of view has driven the modification of the processor architecture to achieve a complete fault coverage. Experimental results, carried out on a specific architecture, by means of a fault injection campaign supported the presented solution.

Acknowledgments

We would like to thank our colleagues, Proff. Fabio Salice and Matteo Sonza Reorda for their helpful discussions.

References

- [1] Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham. Design and evaluation of system-level checks for on-line control flow error detection. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):627–641, 1999.
- [2] P. Bernardi, L. Bolzani, M. Rebaudengo, M. Sonza Reorda, and M. Violante. An integrated approach for increasing the soft-error detection capabilities in SoCs processors. In *Proc. 20th IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems*, pages 445–453, 2005.
- [3] C. Bolchini, A. Miele, L. Pomante, F. Salice, and D. Sciuto. Reliable system co-design: The FIR case study. In *Proc. 19th IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems*, pages 433–441, Cannes, France, 2004.
- [4] C. Bolchini, A. Miele, F. Salice, and D. Sciuto. A model of soft error effects in generic ip processors. In *Proc. 20th IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems*, pages 334–342, 2005.
- [5] C. Bolchini, L. Pomante, F. Salice, and D. Sciuto. Reliable system specification for self-checking datapaths. In *Proc. of the Conf. on Design, Automation and Test in Europe*, pages 1278–1283, Washington, DC, USA, 2005. IEEE Computer Society.
- [6] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, and M. Violante. An FPGA-Based approach for speeding-up fault injection campaigns on safety-critical circuits. *Journal of Electronic Testing: Theory and Applications*, 18(3):261–271, June 2002.
- [7] F. Ziegler et al. Terrestrial cosmic rays and soft errors. *IBM Journal of Research and Development*, 40(1), 1996.
- [8] O. Goloubeva, M. Rebaudengo, M.S. Reorda, and M. Violante. Improved software-based processor control-flow errors detection technique. In *Reliability and Maintainability Symposium, 2005. Proceedings. Annual*, pages 583–589, 2005.
- [9] J. Gaisler. *The LEON2 IEEE-1754 (SPARC V8) Processor*, 2003. <http://www.gaisler.com>.
- [10] M. Rebaudengo, M. Sonza Reorda, M. Torchiano, and M. Violante. Soft-error detection through software fault-tolerance techniques. In *Proc. IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems*, pages 210–218, 1999.