Hybrid Soft Error Detection by means of Infrastructure IP cores

L. Bolzani[1], M. Rebaudengo[2], M. Sonza Reorda[2], F. Vargas[1], M. Violante[2]

[1] Departamento de Engenharia Elétrica,
Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS),
Porto Alegre, Brazil

[2] Politecnico di Torino
Dipartimento di Automatica e Informatica
Torino, Italy

Abstract*

High integration levels coupled with the increased sensitivity to soft errors even at ground level make the task of guaranteeing adequate dependability levels more difficult then ever. In this paper we propose to adopt low-cost infrastructure-intellectual-property (I-IP) cores in conjunction with software-based techniques to perform soft error detection. Experimental results are reported that show the effectiveness of the proposed approach.

1. Introduction

The always increasing number of computer-based safety-critical applications rises serious questions about the methods for guaranteeing sufficient degrees of reliability while still maintaining reasonable costs for design and manufacturing.

As far as microprocessor-based systems are considered, several solutions [1] have been proposed in the past, which can be categorized in two groups: *hardware-based approaches*, relying on custom hardware, and *software-based approaches*, relying on commercial hardware, and exploiting carefully devised software to achieve fault tolerance (or safety).

When the architecture of the microprocessor (μP) that needs to be hardened is fixed, a typical hardware-based fault tolerance solution consists in introducing a special-purpose processor, called *watchdog processor*, that constantly monitors the activities carried out by the μP. As soon as a misbehavior is observed, suitable fault-containment procedures are activated. For a survey about designing watchdog processors readers may refer to [2]. Conversely, when designers have freely access to the μP's internal architecture, fault-tolerant design techniques like information redundancy [3] and component redundancy [1] can be exploited. Although effective from the fault containment point of view, these approaches mandate high area overheads and thus they are appealing only to high-end applications, where cost is a minor issue.

With the adoption of computer-based systems in fields where cost is a major issue (such as the automotive, or the biomedical one), the interest for software-based approaches started increasing significantly. These approaches, like [4], [5] and [6], improve the dependability of computer-based systems by acting only on the software, while the underlying hardware remains unchanged. When such approaches are exploited, commercial-of-the-shelf (COTS) processors are used to run special-crafted versions of the software that allow fault detection and possibly correction. These approaches are able to cope well with both permanent and transient faults, but may introduce significant performance degradations as well code/data memory overhead.

The current design practice, which is seeing the success of System-on-Chips (SoCs) build on top of IP-cores coming from third parties, suggests new approaches to provide fault tolerance (or safety) with reduced overheads.

IP-cores can belong to different categories: apart from those implementing general-purpose functions (e.g., memories, processors, custom logic, etc.), others are now added to SoCs to achieve some specific non-functional goals, such as increasing yield, simplifying debugging, supporting testing, etc. These IP-cores are often referred to as *Infrastructure IP-cores* (I-IPs) [7].

Within this framework, this paper investigates the possibility of exploiting I-IPs for quickly and economically developing safety-critical SoCs. The result of our investigations is a hybrid approach that combines software-based techniques with hardware-based ones [8]. In this way, we combine the benefits of both approaches, i.e., low development cost and low overhead in terms of performance penalties.

* This work has been partially supported by the European Commission through the ALFA II-0086-FI project Tosca.

Our approach exploits code-transformation rules (like those proposed in [4]) to automatically introduce information and operation redundancies to the software portion of the system we need to harden, resulting in a program where all the variables and the operations among them are duplicated. Moreover, an I-IP tailored to the processor running the modified software is used to manage consistency checks on the stored redundant information. The I-IP constantly monitors the processor system bus: as soon as two copies of the same variable are read, it checks whether they coincide, and it activates an error signal in case a mismatch is found.

The main novelty of this paper is in proposing an hybrid hardening technique where information redundancy is implemented in software, while fault detection is performed in hardware.

In order to assess the capabilities of the proposed approach, we developed a prototypical implementation targeting the Intel 8051 microcontroller. Three benchmark programs were hardened according to the software modification rules we adopted, and an I-IP was developed to interact with the Intel 8051 core. The fault-injection experiments we performed showed that the resulting system is able to effectively deal with the detection of transient faults affecting the stored information, while it requires limited overhead in terms of memory occupation and performance degradation. Moreover, the proposed approach allows to easily transform an existing µP core into a safe one, by simply complementing it with the I-IP core in charge of fault detection.

The remainder of the paper is organized as follows. Section 2 presents the approach we developed, while Section 3 reports the experiments we performed to asses its effectiveness. Finally, Section 4 draws some conclusions.

2. Hybrid fault detection technique

This Section describes the hybrid approach we developed for quickly and economically developing reliable safety-critical SoC-based systems. Sub-section 2.1 reports the assumptions we made in developing our approach, which is presented in Sub-sections 2.2.

2.1. Assumptions

We assume the system to be hardened is a SoC which employs a COTS processor and two COTS memory modules (denoted as code memory or CM and data memory or DM). Memory modules are not integrated in the processor and they are not equipped with any fault detection/correction mechanism. CM stores the binary code of the software running on the processor, while DM stores the data the software manipulates. For the sake of this paper, we assume that the processor is not equipped with cache memory. All the internal details about the processor and the memory modules are neglected and they are not supposed to be accessible for hardening purposes. Conversely, the details about the bus interface connecting the processor with the memories are known.

Moreover, we assume that the source code for the software the processor is intended to run is available. In our work we consider only embedded applications where dynamic memory allocation is not exploited.

We developed our approach assuming that, as soon as a fault is detected by means of an ad-hoc I-IP, an error signal is activated and a suitable fault containment procedure is executed. Fault containment is not addressed in the following, and we concentrated on fault detection, only.

Finally, we assume the fault type against which the SoC needs to be hardened is the *Single Event Upset*, which is a soft error resulting from the perturbation of one storage cell caused by ionization [9]. A characteristic of SEUs is that they are random events and thus they may occur at unpredictable times. For example, they may corrupt the content of a processor register during the execution of an instruction. To model the effects of SEUs we exploited the *transient bit flip* fault model, which consists in the modification of the content of a storage cell during program execution. Possible fault locations are thus internal memory cells, flip-flops, bits of user and control registers and even registers usually not accessible through the processor instruction set, and external memories.

In this paper we considered only bit flips in DM memory and internal memory cells, while faults affecting all the other memory elements in the system are neglected.

2.2. The proposed approach

In the following we will describe our approach by firstly addressing the algorithm the processor-based system executes, while neglecting how information and operation duplications are implemented. Then we will describe how the hardened algorithm is implemented, detailing which portion of the hardened algorithm is implemented in software and which one in hardware.

Given an algorithm, the approach we propose mandates the execution of the following three steps:

- Every variable used by the algorithm is duplicated
- Every computation in the algorithm is duplicated
- During the execution of the hardened algorithm, consistency checks are performed each time variables are read. Consistency checks verify whether faults affected one of the two copies of each variable. If a mismatch is found it means that a fault is *detected*, and in this case a proper fault-containment procedure is activated.

The last step is crucial, since it must be performed after every read operation, and it is therefore potentially rather time consuming. A typical algorithm is indeed composed of several basic blocks, most of them being loops, where variables are read and modified. Since at each loop's iteration all the read variables are checked for consistency, the execution time of the algorithm may be drastically reduced by consistency checks, whose execution time may overcome that for repeating twice all the computations.

For this reason, we propose to implement our hardening approach partly in software and partly in hardware: duplication of variables and computations is implemented in software as already proposed by similar techniques [4], while consistency checks are implemented in hardware. An ad hoc I-IP, called *Cerberus*, is exploited for this purpose, which works in close symbiosis with the COTS processor the SoC embeds. By constantly monitoring the processor bus, Cerberus continuously checks whether read variables are consistent. When a mismatch is found a proper signal is raised (e.g., an interrupt), and a suitable fault-containment procedure is activated. The architecture of the resulting SoC is shown in Figure 1.

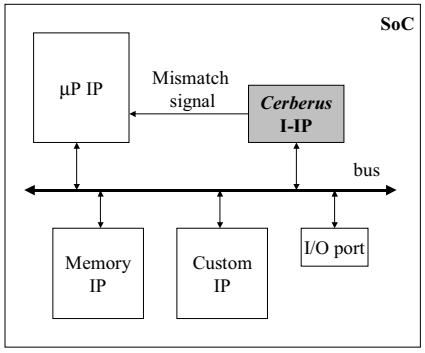

Fig. 1. SoC architecture including an I-IP for soft error detection

The proposed solution presents some significant advantages over a purely software solution. The time overhead is greatly reduced, since all the checks performed after read operations are now performed in hardware. It is much more efficient in terms of area overhead than a purely hardware approach, since the I-IP has a negligible cost (especially when compared to the cost of the processor).

Moreover, the cost for its adoption is relatively low: the same I-IP can be exploited each time a given processor is used, and its integration in the SoC is relatively straightforward. The memory overhead (in terms of code size) is also reduced, due to the same reason.

2.3. Software hardening

The first two steps described in Section 2.2 are implemented by first translating the algorithm in C code and then by applying the following code-transformation rules [4]:

R1. Every variable in the code is duplicated;

R2. Every computation among variables is duplicated.

To give the reader with a flavor of the transformation rules the method is based on, Figure 2 shows how a simple piece of code is transformed.

Being consistency checks performed in hardware, we do not need to add any other instruction to the hardened code.

Original code	*Hardened code*

`int a, b;`
`...`
`a = b + 5;`
	`int a1, b1;`
	`int a2, b2;`
	`...`
	`a1 = b1 + 5;`
	`a2 = b2 + 5;`

Fig. 2. Example of hardened-oriented code transformation

To keep Cerberus's architecture as simple as possible, variables are ordered in DM in such a way that the data segment of the program is divided in two portions, as shown in Figure 3. The upper portion contains the first replica of each variable (for example `a1` and `b1` of Figure 2), while the lower one stores the second replica (`a2` and `b2` of Figure 2). Variables' ordering can be easily obtained through the options of the adopted C compiler.

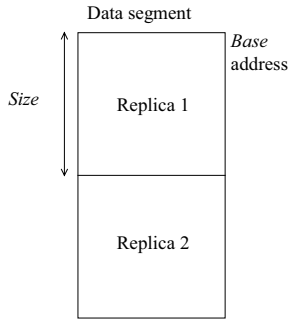

Fig. 3. Data segment of the hardened program

This assumption about the variables' ordering makes particularly simple the identification of the replicas that need to be checked starting from the address of the read variable (obtained at run time by monitoring the processor bus), the *Base* address (corresponding to the beginning of the data segment) and *Size* (corresponding to the size of

one set of replicated variables) of Figure 3, which are obtained at compile time.

2.4. Cerberus' architecture

Cerberus is the I-IP in charge of monitoring the activity on the processor bus during program execution. Each time a variable is read, Cerberus identifies the bus cycles during which the two replicas of the variable travel over the bus. As soon as both replicas have been observed, Cerberus checks whether they hold the same value, and in case a mismatch is found (or the second replica is not accessed), it signals the occurrence of an error.

Cerberus is partitioned in three modules, as depicted in Figure 4: Bus Interface Logic, Consistency Check Logic, and CAM Memory.

The *Bus Interface Logic* implements the interface needed for accessing to the processor bus. It decodes the bus cycles being executed and in case of read or write cycles to the memory, it samples the address (*adx*) and the value (*data*) on the bus. Sampled address and value are then forwarded to the Consistency Check Logic.

The *Consistency Check Logic* implements the consistency checks needed for verifying whether any data stored in the data memory has been affected by SEUs. For this purpose, as soon as a new couple (*adx*, *data*) is available and the processor is executing a read cycle, it executes the following steps:

- Starting from *adx* the equation 1 is used to compute the associated key value $k(adx)$:

$$k(adx) = \begin{cases} adx \Leftrightarrow adx < Base + Size \\ adx - Size \Leftrightarrow adx \geq Base + Size \end{cases} \quad \text{(Eq. 1)}$$

- It looks for $k(adx)$ in the CAM Memory;
- If $k(adx)$ is not found, it is stored in the CAM Memory along with the associated *data*;
- If $k(adx)$ is found, it compares the stored data coming from the CAM Memory, $data^*$, with *data*. If the two values do not match the Mismatch signal is raised to signal that a fault has been detected. Please note that with this approach Cerberus may check the consistency of the *same replica* that is read twice in a row, and that of the *two replicas* of the same variable that are read once.

Conversely, in case the processor is executing a write cycle, the following operations are performed:

- It looks for $k(adx)$, computed by means of equation 1, in the CAM Memory;
- If $k(adx)$ is found, it removes $k(adx)$ and the associated data from the CAM Memory.

Finally, the *CAM Memory* is a memory used to store one replica of each read variable. In order to keep consistency checks as fast as possible, this module is implemented as a content-addressable memory.

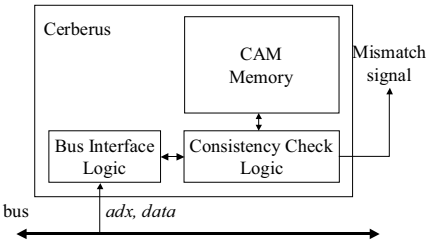

Fig. 4. Architecture of the developed I-IP

Cerberus can be easily adapted to different processors: both the CAM Memory and Consistency Check Logic are parametric and can thus be easily adapted to different address and data sizes. Only the Bus Interface Logic needs to be reworked for adapting to the read and write cycles implemented by different processors.

3. Experimental results

To assess the effectiveness of our hybrid approach, we developed a prototypical implementation assuming the SoC we need to harden is based on the Intel 8051 controller. For this purpose, we modeled Cerberus's behavior in VHDL language, obtaining about 700 lines of code, and we connected it with a soft-core implementing the Intel 8051.

To measure its fault-detection capabilities, we performed several fault-injection campaigns whose results are reported in Section 3.1. We then evaluated its overhead with respect to an un-hardened system; the obtained results are reported in Section 3.2.

3.1. Fault-detection analysis

During our experiments we considered three benchmark programs that are inspired to those in the EEMBC Automotive/industrial suite [10]:

- *Matrix*: it computes the product of two 3x3 matrices.
- *Ellipf*: it computes an Elliptic filters over a set of 6 samples.
- *FIR*: it computes an 8-tap Finite Input Response filter over a set of 16 samples.

During our experiments we considered three different system implementations:

- A first version (*Plain*) where the Intel 8051 controller runs the plain version of the considered benchmarks, and where no hardware or software fault detection techniques are exploited.
- A second one (*Soft*) based on the purely software approach described in [4], where data and operations duplications, as well as consistency checks are all performed in software. For the sake of this paper,

hardening techniques intended for preserving the expected execution flow are not exploited. As a result, some faults that alter the expected execution flow, for example by modifying the value of loop's indexes, are expected not to be detected.

- A final one (*Hybrid*) where the Intel 8051 controller is equipped with our hybrid approach: Cerberus is exploited in conjunction with software modification rules for duplicating data and operations.

After developing the three system implementations, we performed several fault-injection campaigns: 10,000 faults were initially injected in the data segment and processor user-accessible registers for implementation *Plain*. Then, a number of faults that is proportional to the data and performance overheads introduced by the hardened implementations are injected in *Soft* and *Hybrid* versions, respectively. This criteria was adopted to take into account that, being the hardened programs larger than the original one, and lasting for a longer execution time, they are more sensitive to the appearance of SEUs than the original one.

Prog.	Ver.	Silent [#]	Time-out [#]	Detected [#]	Wrong Answer [#]
Matrix	*Plain*	8,702	21	0	1,277
	Soft	46,543	144	22,021	0
	Hybrid	23,464	131	10,007	13
Ellipf	*Plain*	9,536	7	0	457
	Soft	67,567	81	22,011	0
	Hybrid	26,885	0	6,697	3
FIR	*Plain*	8,568	6	0	1,426
	Soft	51,393	39	25,635	8
	Hybrid	23,449	45	10,973	7

Table 1. Results of fault-injection campaigns

The results coming from the fault-injection campaigns we performed are summarized in Table 1. Faults were injected by exploiting the fault-injection environment described in [11]. Fault location and fault-injection time were randomly selected among the locations in DM and the program duration. Fault effects were classified as follows:

- *Silent*: the fault did not modify the execution of the program. The obtained output results coincides with those of the program running when any fault is injected;
- *Time-out*: the fault modified program execution in such a way that the processor entered in an end-less loop, and it was not able to provide any results.
- *Detected*: the fault modified the program execution, but the implemented fault-confinement technique was able to detect it.
- *Wrong Answer*: the fault modified the program execution, and its output results differ from the expected ones. These faults are the most critical ones, and their number should be minimized by the adopted detection mechanism.

As the reader can observe from Table 1, the hybrid approach is able to effectively reduce the number of *Wrong Answers* for all the considered benchmarks with respect to the unhardened version. Its fault-detection capability is also comparable to that of a purely software approach, and sometimes it is even slightly better. This latter point can be explained by observing that the hybrid approach is able to detect some faults modifying the code execution flow, which escape the detection mechanisms embedded in the software hardened version.

3.2. Overhead analysis

The hybrid approach we propose encompasses three types of overheads: an area overhead related to the adoption of Cerberus, a memory overhead due to data and operation duplications, and a performance overhead due to operation duplications.

In order to quantify the area occupation of Cerberus, we synthesized it by exploiting Synopsys Design Analyzer and a technology library developed at our institution. Cerberus was configured to interact with the internal memory bus of the Intel 8051 controller we adopted, and it was configured with a CAM Memory able to store 16 entries. The gate-level model of Cerberus accounts for 1,636 gates and 288 flip-flops, while the gate-level model of the considered Intel 8051 controller accounts for 10,723 gates and 19,757 flip-flops (data and code memories were implemented as static memories). The percent area overhead Cerberus introduces is expected to further decrease when increasing the size of the companion processor.

To quantify the memory and performance overheads we measured the memory occupation of the programs that were hardened according to the hybrid approach, and then we compared it with that of the same programs hardened according to [4]. As a reference, we also measured the area occupation and program duration of the original programs. In Table 3 we reported the figures we recorded. Memory occupation was reported as number of bytes in the data and code segments, while duration was reported as number of clock cycles for program completion.

As the reader can observe, with the hybrid approach the memory occupation increases by a factor of about 1.7 and the program duration of about 2.0. Conversely, when a full software approach is used as that described in [4], the memory occupation increases by a factor of about 3.7, while the program duration of about 4.0.

When this figures are coupled with those referring to area overhead and fault detection capabilities, we can conclude that the hybrid approach is able to improve effectively the dependability of a computer-based system with limited area overhead, memory increase and performance degradation.

Prog.	Ver.	Memory Occupation [# bytes]	Program Duration [# clocks]
Matrix	*Plain*	246	29,715
	Soft	720	102,084
	Hybrid	438	49,945
Ellipf	*Plain*	359	16,268
	Soft	1,755	72,929
	Hybrid	667	27,318
FIR	*Plain*	228	43,434
	Soft	793	167,381
	Hybrid	544	74,867

Table 3. Memory occupation and program duration.

4. Conclusions

We presented an hybrid approach that exploits software-based techniques in collaboration with an infrastructure IP-core to perform error detection in processor-based SOCs. Software techniques are used to enrich the application running on SOCs with data and operations redundancies. Meanwhile, an I-IP is used to perform consistency checks as soon as the information the program manipulates is accessed.

To assess the soundness of our approach, we implemented it in a prototype based on the Intel 8051 controller. Three benchmarks were considered, and different implementations developed: an un-hardened one, one hardened according to a purely software approach, and one hardened according to our hybrid approach.

The fault-injection experiments we performed gave evidences of the effectiveness of the approach as far as its fault detection capability was concerned. Moreover, analysis of the overhead with respect to the un-hardened programs showed that the hybrid approach was able to meet high dependability levels with limited area, memory and performance overheads.

References

[1] D.K. Pradhan, Fault-Tolerant Computer System Design, Prentice Hall, 1996

[2] A. Mahmood, E. J. McCluskey, "Concurrent error detection using watchdog processors-a survey", IEEE Transaction on Computers, Vol. 37, No. 2, February 1988, pp. 160-174

[3] M. Pflanz, H. T. Vierhaus, "Online check and recovery techniques for dependable embedded processors", IEEE Micro, Vol. 21, No. 5, Sept.-Oct. 2001, pp. 24-40

[4] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. Sonza Reorda, M. Violante, "Experimentally evaluating an automatic approach for generating safety-criticil software with respect to transient errors", IEEE Transaction on Nuclear Science, Vol. 47, No. 6, December 2000, pp. 2231-2236

[5] N. Oh, P. P. Shirvani, E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors", IEEE Transactions on Reliability, Vol. 51, No. 1, March 2002, pp. 63-75

[6] N. Oh, S. Mitra, E. J. McCluskey, "Error detection by diverse data and duplicated instructions", IEEE Transactions on Computers, Vol. 51, No. 2, February 2002, pp. 180-199

[7] 1st International Workshop on Infrastructure IP, Held in conjunction with the IEEE International Test Conference, October 2004

[8] L. Bolzani, M. Rebaudengo, M. Sonza Reorda, F. Vargas, M. Violante, "An Infrastructure IP for Soft Error Detection", included in the Informal Compendium of Papers of the 5th Latin-American Test Workshop

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

[10] www.eembc.org

[11] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, M. Violante, "An FPGA-based approach for speeding-up Fault Injection campaigns on safety-critical circuits", Journal of Electronic Testing:Theory and Applications, Vol. 18, No. 3, June 2002, pp. 261-271