On-line Detection of Control-Flow Errors in SoCs by means of an Infrastructure IP core

P. Bernardi[2], L. Bolzani[2], 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
{fvargas}@ee.pucrs.br

[2] *Politecnico di Torino*
Dipartimento di Automatica e Informatica
Torino, Italy
{paolo.bernardi, leticia.veirasbolzani, maurizio.rebaudengo, matteo.sonzareorda, massimo.violante}@polito.it

Abstract[1]

In sub-micron technology circuits high integration levels coupled with the increased sensitivity to soft errors even at ground level make the task of guaranteeing systems' dependability more difficult than ever. In this paper we present a new approach to detect control-flow errors by exploiting a low-cost Infrastructure Intellectual Property (I-IP) core that works in cooperation with software-based techniques. The proposed approach is particularly suited when the system to be hardened is implemented as a System-on-Chip (SoC), since the I-IP can be added easily and it is independent on the application. Experimental results are reported showing the effectiveness of the proposed approach.

1. Introduction

The always-increasing number of computer-based safety-critical applications raises 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) 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 any 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 free access to the μP's internal architecture, fault-tolerant design techniques such as 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 for 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 ([4][5][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-Off-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 as code/data memory overhead.

The current design practice, which is seeing the success of System-on-Chips (SoCs) built on top of IP-cores coming from third parties, suggests new

[1] This work was partially supported by the European Union through the ALFA project TOSCA.

Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)
0-7695-2282-3/05 $20.00 © 2005 IEEE

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 the above 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.

Our approach combines a software hardening technique, which is used to track the execution flow of programs, with a low-cost I-IP that is in charge of continuously controlling that the program flow is correctly executed. Execution flow is tracked by inserting (at suitable points within the code) ad hoc instructions able to inform the I-IP about which part of the application is currently being executed. More in detail, these instructions send information to the I-IP each time the execution flow enters or leaves a basic block [9]. In response, the I-IP checks whether the program's evolution is compatible with the expected behavior, and activates a suitable error handling procedure as soon as an error is detected.

The approach proposed in this paper shares some ideas with the one proposed in [8]: however, the latter was targeted to detect faults affecting the data used by programs, while this one targets faults modifying the execution flow of programs. Consequently, the functions and the architecture of the I-IPs proposed in the two papers are different.

The proposed I-IP is very simple. It does not embed any information about the program under execution: it merely performs consistency checks on the basis of information the running program sends to the I-IP. On the one hand, this approach reduces the performance overhead that would affect hardened programs in the case the consistency checks are performed completely in software. On the other hand, this approach makes the I-IP very general: once the I-IP is tailored to a specific processor, it can be used for every application the processor executes.

The possibility of applying automatically the proposed hardening technique is another advantage of the approach. Basic blocks within programs' source code can be identified resorting to already available automatic techniques [9]. Moreover, the obtained information can also be used to automatically obtain the hardened version of programs.

In order to assess the capabilities of the proposed approach, we developed a prototypical implementation targeting the Intel 8051 microcontroller. Four benchmark programs were hardened according to the approach we developed, and an I-IP dedicated to the Intel 8051 was developed. The fault-injection experiments we performed showed that the resulting system is able to effectively deal with the detection of transient faults affecting programs' control flow, while it requires limited overhead in terms of memory occupation and performance degradation. Moreover, the proposed approach allows transforming easily 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 an overview of the already available techniques coping with control-flow check. Section 3 presents the approach we developed, while section 4 reports the experiments we performed to assess its effectiveness. Finally, Section 5 draws some conclusions.

2. Previous works

In this section we propose a survey of previously proposed error-detection techniques for microprocessor-based systems. These techniques can be organized in two broad categories: software-implemented techniques, which exploit purely software detection mechanisms, and hardware-based ones, which exploit additional hardware.

2.1. Software-based techniques

Software-based techniques exploit the concepts of information, operation, and time redundancy to detect the occurrence of errors during program execution.

Among the past thirty years many techniques have been developed. The older techniques obtain a dependable software-based system by replicating the execution of the program, and by voting among the results produced by each replica (e.g., Recovery Blocks [10], and N-Version Programming [11]). Although effective, these approaches rely on software designers for their implementations: designers are in charge of devising how to replicate the program, and how to implement the voting mechanism that best fits the application the program implements. These processes are in general not automated, and thus they are error prone.

More recent techniques (like [12]) harden programs against errors by introducing some control instructions.

These techniques simplify the task for software designers, since some of them can be applied automatically to the software that to be harden. However, some of these techniques are not general, since they are dedicated to harden a certain class of applications, only.

In the past five years some techniques have been developed that can be applied automatically to the source code of a program thus simplifying the task for software developers: the software is indeed hardened by construction, and the development costs can be reduced significantly. Moreover, the most recently proposed techniques are general, and thus they can be applied to a wide range of applications.

Those techniques aiming at detecting the effects of faults that modify the expected program's execution flow are known as control-flow checking techniques. These techniques are based on partitioning the program's code into basic blocks [9]. A basic block is a sequence of consecutive instructions in which, in absence of faults, the flow of control always enters at the beginning and leaves at the end. This means that a basic block does not contain any instruction that may change the control flow, such as jump, branch or call instructions, except for the last one, possibly. Furthermore, no instructions in the basic block can be the destination of a branch, jump or call instruction, except for the first one, possibly.

A program P can be represented with a graph composed of a set of nodes V and a set of edges E, P={V, E} [13], where V={v_1, …,v_i,…,v_n} and E={e_1,…,e_i,…,e_m}. Each node v_i represents a basic block and each edge e_i represents the branch $br_{i,j}$ from v_i to v_j. The edges $br_{i,j}$ are not necessarily explicit branch instructions, but they also can represent jumps, subroutine calls, return instructions.

Considering the Program Graph P={V, E}, for each node v_i it is possible to define suc(v_i) as the set of nodes successor of v_i and pred(v_i) as the set of nodes predecessor of v_i. A node v_j belongs to suc(v_i) if and only if $br_{i,j}$ is included in E. Similarly, v_j belongs to pred(v_i) if and only if $br_{j,i}$ is included in E.

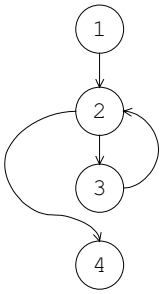
Considering a program represented by its Program Graph P={V, E}, then during the execution of P, $br_{i,j}$ is illegal if $br_{i,j}$ is not included in E. This illegal branch indicates a control-flow error, which can be caused by transient or permanent faults. Two types of control-flow errors can be defined:

1) Inter-block faults, that consist in illegal branches $br_{i,j}$, where $br_{i,j} \notin E$, and $i \neq j$, i.e., branches between two different basic blocks.
2) Intra-block faults, that consist in illegal branches $br_{i,i}$, where $br_{i,i} \notin E$, i.e., branches within the same basic blocks.

As an example, we consider the fragment of a sample routine shown in figure Fig. 1, where the basic blocks are also numbered in a), and the corresponding Program Graph is shown in b).

```
1: x = 1;
   y = 5;
   i = 0;
2: while( i < 5 ) {
3:   z = x+i*y;
     i = i+1;
   }
4: i = 2*z;
```

a)



b)

Fig. 1. Source code and program graph example.

The most important solutions proposed in the literature are the techniques called *Enhanced Control Flow Checking using Assertions* (ECCA) [14] and *Control Flow Checking by Software Signatures* (CFCSS) [15].

ECCA assigns a unique prime number identifier to each basic block of a program. A global integer variable is added to check the control flow correct execution. This variable is dynamically updated during the execution. Two lines of code are asserted into each block:

1) A test assertion is executed at the beginning of the block for checking if the previous basic block is permissible, according to the Program Graph; a divide by zero error identifies the control flow error,
2) A set assignment is executed at the end of the block for updating the identifier taking into account the whole set of possible next basic blocks.

ECCA is able to detect all the single inter-block control-flow errors. It is not able to detect intra-block control-flow errors, neither faults that cause an incorrect decision on a conditional branch.

CFCSS assigns a unique signature s_i to each basic block. A global variable (called G) contains the run-time signature. In absence of errors, G contains the signature associated to the current basic block. G is initialized with the signature of the first block of the program.

At the beginning of the basic block an additional instruction computes the signature of the destination block from the signature of the source block: G is updated with the EXOR function between the signature of the current node and the destination node. If the

control can enter from multiple blocks, then an adjusting signature is assigned in each source block and used in the destination block to compute the signature.

As a limitation, CFCSS cannot cover control flow errors if multiple nodes share multiple nodes as their destination nodes. As described in [15].

2.2. Hardware-based techniques

Hardware-based techniques exploit special-purpose hardware modules, called *watchdog processors* [2], to monitor the control-flow of programs, as well as memory accesses. A watchdog processor is a simple processor that detects errors by monitoring the behaviour of a main processor, according to the architecture of Figure 2.

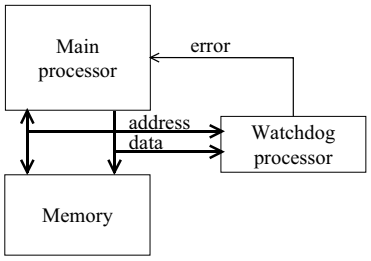


Fig. 2. The basic architecture of a system hardened through a watchdog processor

A watchdog processor can perform three types of monitoring operations.

1) *Memory-accesses checks*, which consist in monitoring for unexpected memory access executed by the main processor. As an example, an approach is proposed in [16] where the watchdog processor knows at each time during program execution which portion of the program's data and code can be accessed. In case the main processor executes an unexpected access, an error signal is activated.
2) *Consistency checks* of variables' contents, which consist in controlling if the value a variable holds is plausible. By exploiting the knowledge about the task performed by the hardened program, watchdog processors can validate each value the processor writes or reads through range checks, or by exploiting known relationships among variables [17].
3) *Control-flow checks*, which consists in controlling whether all the taken branches are consistent with the Program Graph [18].

As far as the control-flow check is considered, two types of watchdog processor may be envisioned:

1) *Active watchdog processor*. This type of watchdog executes a program concurrently with the main processor. The watchdog' program has a Program Graph that is homomorphic to the main processor one. During program execution, the watchdog continuously checks whether its program evolves as that executed by the main processor [2][19]. This solution introduces minimal overhead in the program executed by the main processor; however, the area overhead needed for implementing the watchdog processor can be non-negligible.
2) *Passive watchdog processor*. This type of watchdog does not execute any program; conversely, it computes a signature by observing the main processor' bus. Moreover, it performs consistency checks each time the main program enters/leaves a basic block within the Program Graph. A cost-effective implementation is described in [20], where a watchdog processor observes the instructions the main processor executes, and computes a run-time signature. Moreover, the code running on the main processor is modified in such a way that when entering a basic-block, an instruction is issued to the watchdog processor with a pre-calculated signature, while the main processor executes a NOP instruction. The watchdog processor compares the received pre-computed signature with the one computed at run-time, and it issues an error signal in case of mismatch. An alternative approach is proposed in [21] where the watchdog processor computes a run-time signature on the basis of the addresses of the instructions the main processor fetches. Passive watchdog processors are potentially simpler than active ones, since they do not need to embed the Program Graph, and since they perform simpler operations: signature computation can be demanded to Linear Feed-Back Shift-Registers (LFSRs), and consistency checks to comparators. However, an overhead is introduced in the monitored program: instructions are indeed needed for communicating with the watchdog.

3. Hybrid fault detection technique

This section describes the hybrid approach we developed for quickly and economically implementing reliable safety-critical SoC-based systems. Sub-section 3.1 reports the assumptions we made in developing our

approach, which is presented in Sub-sections 3.2 and 3.3.

3.1. Assumptions

We assume the system to be hardened is a SoC that employs a (possibly pipelined) COTS processor; all the internal details about the processor are neglected and they are not supposed to be accessible for hardening purposes. Moreover, we assume that the source code for the application program 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 (and possibly correction) procedure is executed. This procedure is not addressed in the following, where we concentrated on fault detection, only.

Concerning the fault model, we adopted the control-flow error as the reference fault model for developing our hardening approach.

3.2. The proposed approach

Given a program to be hardened, our approach exploits a modified version of the program's source code in combination with an ad-hoc developed I-IP.

The control flow of the program is tracked by inserting at suitable points within its code instructions to inform the I-IPs about which part of the application is currently being executed. These instructions send information to the I-IP each time the control flow enters or leaves a basic block. The I-IP constantly monitors the bus of the processor running the modified program. As soon as it receives updated information about the application it checks whether the program's evolution is compatible with the expected behavior, and it activates a suitable error handling procedure as soon as an error is detected.

Our approach is similar to that based on passive watchdog processors: in both approaches the software indeed communicates with special-purpose hardware modules. However, our approach is innovative since, as detailed in the following, the I-IP we devised is much simpler than a watchdog. Our I-IP constantly monitors the processor bus, but it operates only when it receives a communication from the hardened software: power consumption is thus minimized. Moreover, the I-IP implements a simple Boolean function, and thus its area is very limited.

In our approach, we check the programs' control flow by using a run-time signature B_i associated with the basic block v_i the processor is executing. The value B_i associated with v_i is unique and is defined at the compile time.

The following assertions (first presented in [22]) are introduced into each basic block v_i to perform the control-flow check:

- A *test assertion* controls the signature of the basic block v_j from which the control flow has reached v_i. This operation is performed by checking if v_j belongs to pred(v_i), according to the program graph.
- A *set assignment* updates the signature setting it to the value B_i associated to v_i. This operation is implemented as $B_i = (B_j \ \& \ M_1) \oplus M_2$, where M_1 represents a constant mask depending on the signatures of the nodes belonging to pred(v_i), while M_2 represents a constant mask depending both on the signature of the current node and those of the nodes belonging to pred(v_i). The appropriate values of M_1 and M_2 must be defined, in order to obtain that the set operation modifies the value of the signature into the proper one, e.g., the signature B_i of the current node v_i. The computation of the values B_i for each basic block, of M_1, and M_2 are performed at compile time according to the algorithm presented in [22].

As an example of how a program is hardened according to the approach we adopted, let us consider the example shown in Figure 3, where $\sum$ is the program's run-time signature, lines 3 and 4 implement the test assertion, and line 10 implements the set one.

```
01: while(k1<DIM)
02: {
03:     if( ∑!=BB1 && ∑!=BB2 )
04:       //Error detected
05:     A1 = matrixA1[i1][k1];
06:     B1 = matrixB1[k1][j1];
07:     C1 += A1*B1;
08:     matrixC1[i1][j1] = C1;
09:     k1++;
10:     ∑=(∑ & M1_BB2)^M2_BB2;
11: }
```

Fig. 3. A code fragment hardened according to the approach presented in [22]

```
01: while(k1<DIM)
02: {
03:     IIPtest( BB1 );
04:     IIPtest( BB2 );
05:     A1 = matrixA1[i1][k1];
06:     B1 = matrixB1[k1][j1];
07:     C1 += A1*B1;
08:     matrixC1[i1][j1] = C1;
09:     k1++;
10:     IIPset( BB2 );
11: }
```

Fig. 4. The hardened code that communicates with the I-IP.

As the reader can observe, the method implies very limited modifications of applications' original code, and it has already been proved effective in terms of memory overhead, and fault detection [22]. However, it may introduce performance penalties due to the additional instructions needed to implement the test and set assertions.

For this reason, we propose to implement our hardening approach partly in software and partly in hardware by using a I-IP (called *Pandora*), which is described in the following.

Given a basic block v_i, the application's source code is modified as follows:

- For each $v_j \in pred(v_i)$, a statement `IIPtest(`B_j`)` is added to the beginning of v_i. As explained in sub-section 3.3, the statement sends the signature B_j associated with v_j to Pandora. Upon receiving the value, Pandora checks if B_j differs from the current value of the program's signature Pandora stores. In case a mismatch is found an *error* flag is set to TRUE.
- A statement IIPset(Bi) is added to the very end of vi. The statement sends to Pandora the new value for the signature of the basic block. Upon receiving the value, in case the error flag is TRUE, a Mismatch signal is activated; otherwise, Pandora computes the values of masks M1 and M2 for vi on the basis of the received values Bj and Bi, and then updates the program's signature.

Figure 4 shows the previous example hardened according to our approach. As the reader can observe, the code is now much simpler than that in Figure 3; moreover, the application no longer implements the computations needed by the test and set assertions. The time overhead the hybrid approach introduces is thus expected to be much lower than that of the purely software implementation reported in Figure 3.

3.3. Pandora's architecture

The architecture of a SoC hardened according to our approach is shown in Figure 5.

Pandora is partitioned in two modules, as depicted in Figure 6: Bus Interface Logic, and Consistency Check Logic.

The *Bus Interface Logic* implements the interface needed for communicating with the processor bus. In its current implementation Pandora is an isolated-I/O device, and it is connected to the processor's bus devoted to communicating with I/O devices.

The *Consistency Check Logic* implements the test and set assertions needed for verifying whether any control flow error affected the application's expected behavior, and to inform the system through the Mismatch signal that an error has been detected. It is internally provided with both the circuitry to store and update the current signature at each `IIPset()` operation and with the circuitry to check the current flow and calculate at run-time the needed masks when `IIPtest()` operations are executed.

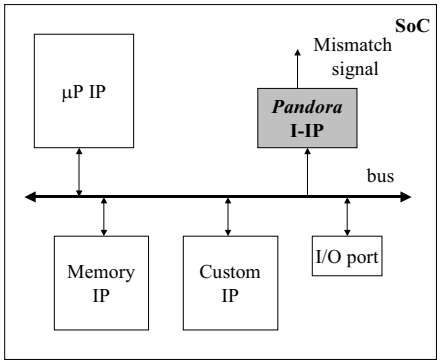


Fig. 5. Architecture of the hardened SOC

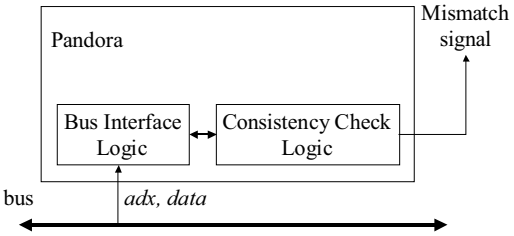


Fig. 6. Architecture of the proposed I-IP

Pandora can be easily adapted to different processors: only the Bus Interface Logic needs to be reworked for adapting Pandora to the read and write cycles implemented by different processors.

4. 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 Pandora's behavior in VHDL language, obtaining about 215 lines of code, and we connected it with an Intel 8051. Communications between Pandora and the processor takes place through the Intel 8051 input/output ports.

To measure Pandora's fault-detection capabilities, we performed several fault-injection campaigns whose results are reported in Section 4.1. We then evaluated

the overhead with respect to an un-hardened system; the obtained results are reported in Section 4.2.

4.1. Fault-detection analysis

To measure the fault-detection capabilities of our approach we performed several fault injection campaigns whose results will be outlined in this paragraph; we assumed that the fault type against which the SoC needs to be hardened is the Single Event Upset (SEU), which is a soft error resulting from the perturbation of one storage cell caused for example by ionization [23][24]. A characteristic of SEUs is that they are random events and thus they may occur at unpredictable times. To model the effects of SEUs, we exploited the transient single bit-flip fault model, which consists in the modification of the content of a single storage cell during program execution.

For the purpose of this paper we are mainly interested in faults affecting the execution control flow; for this reason we considered (as done in [18]) a subset of all possible locations. More in particular, given a branch instruction *J* executed at time *T*, we computed all the possible control-flow errors affecting *J*, i.e., all the bit-flip faults, each originated at time *T* and located in one of the memory bits of the instruction's displacement field. For the considered programs we identified all the branch instructions they execute, as well as all the execution times. We then applied the aforementioned procedure to compute the list of faults, which were injected resorting to the tool presented in [25].

We underline that our fault model is independent on the physical location of the bit affected by the SEU, which can be stored in the external memory, in the cache, or in an internal processor register (e.g., the Instruction Register).

During our experiments we considered four benchmark programs that are inspired to those in the EEMBC Automotive/industrial suite [26]:

- *Matrix*: it computes the product of two 3x3 matrices
- *Ellipf*: it implements an Elliptic filter working on a set of 6 samples
- *FIR*: it computes an 8-tap Finite Input Response filter over a set of 16 samples
- *Viterbi*: it computes the Viterbi encoding for a 4-byte message.

During our experiments we considered four different system implementations:

- *Plain*: the plain version of the considered benchmarks; no hardware or software fault detection techniques are exploited.
- *Pandora*: the hardened versions of the benchmarks obtained using the approach here described.
- *ECCA*: the hardened versions of the benchmarks obtained using the purely software approach described in [14].
- *CFCSS*: the hardened versions of the benchmarks obtained using the purely software approach described in [15].

Similarly to [15], for each program we computed the set of control-flow errors, we injected them, and we compared the obtained results.

To summarize the hardening capabilities of the considered approaches, as well as the impact of control-flow errors in the unhardened version of the programs, we reported in table 1 the percentage of escaped faults, i.e., the percentage of the injected faults for which the program's results were different than the expected ones.

Program	Plain [%]	Pandora [%]	ECCA [%]	CFCSS [%]
Matrix	9.78	0.18	0.99	4.88
Ellipf	20.83	0.00	2.38	14.29
FIR	5.64	0.00	2.12	4.49
Viterbi	21.06	4.89	6.33	17.48

Table 1. Fault injection results

As the reader can observe, our approach is particularly effective for all but the Viterbi benchmark, where it is however better than the other approaches.

When analyzing the gathered results, we come to the following conclusions.

Purely software approaches, such as CFCSS and ECCA, may introduce additional branches to programs' control flow graph. The C compiler may indeed translate some of the C instructions needed to implement CFCSS and ECCA as sequences of assembly-level instructions containing *new branches*. As a result, the actual programs' control flow graph would contain more braches than those inferred by analyzing the programs' source code. Therefore, the new branches would not be protected against control-flow errors, and thus some faults may escape CFCSS's and ECCA's detection capabilities. Conversely, Pandora does not introduce any new branch, and therefore it is able to harden each branch in the program's control flow graph.

Malicious faults exist that alter the execution flow of a program in such a way that the taken branch is consistent to the program's control flow graph, but the branch is taken at un-expected time, or it is not consistent with the information the program is elaborating. Let us consider as an example the line 1 of the code fragment in Figure 4. Depending on the code's size, a bit flip in the displacement field of the branches used to implement the `while` statement may provoke a branch to the loops' exit before the loop's termination condition is met. As a result, the loops' is terminated earlier than expected. However, the original program's

control flow is preserved, and thus this type of faults escapes the adopted detection mechanism, asking for complementary techniques to deal with them. This is the case for the Matrix and Viterbi benchmarks that are particularly loop-intensive.

4.2. Overhead analysis

The hybrid approach we propose encompasses three types of overheads: a silicon overhead related to the adoption of Pandora, and memory and performance overheads due to the introduced modifications in the application's source code.

Program	Plain [byte]	Pandora [byte]	ECCA [byte]	CFCSS [byte]
Matrix	223	385	902	456
Ellipf	303	361	640	347
FIR	194	364	701	320
Viterbi	436	707	1,115	725

Table 2. Memory occupation

Program	Plain [cycle]	Pandora [cycle]	ECCA [cycle]	CFCSS [cycle]
Matrix	31,211	41,462	102,356	43,791
Ellipf	16,268	17,815	25,635	17,611
FIR	43,434	71,994	153,458	57,357
Viterbi	286,364	328,150	349,111	314,244

Table 3. Execution time

In order to quantify the area occupation of Pandora, we synthesized it by exploiting Synopsys Design Analyzer and a technology library internally developed at Politecnico di Torino. Pandora was configured to interact with the internal memory bus of the Intel 8051 controller we adopted. The gate-level model of Pandora accounts for 992 gates, while the gate-level model of the considered Intel 8051 controller accounts for 30,480 gates (data and code memories were implemented as static memories). The percent area overhead Pandora introduces is thus about 3.2%, and it 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 and execution time 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 purely software approaches. Tables 2 and 3 report the figures we attained. As the reader can observe, Pandora introduces memory and performance overheads that are comparable to those of CFCSS, although slightly higher, while its error detection capability outperforms CFCSS. Moreover, Pandora is far more effective than ECCA when both overheads are considered.

5. Conclusions

In this paper we presented a hybrid approach to cope with control-flow errors. The proposed approach presents some significant advantages over purely software solutions. The time overhead is greatly reduced since the most time-consuming operations needed to control programs' execution flow are implemented 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). Finally, the cost for its adoption is relatively low: the same I-IP can be exploited each time the same processor is used, and its integration in the SoC is relatively straightforward. The reported experimental results show the effectiveness of the developed approach, which performs better both in terms of introduced overheads, and error-detection capabilities than already proposed ones.

The method presented in this paper is complementary with respect to the approach proposed in [8] which was used for data checking instead of control flow checking. The two I-IPs own a common interface to the bus able to spy traveling data and can be easily integrated to obtain a better coverage for transient faults affecting the inspected system.

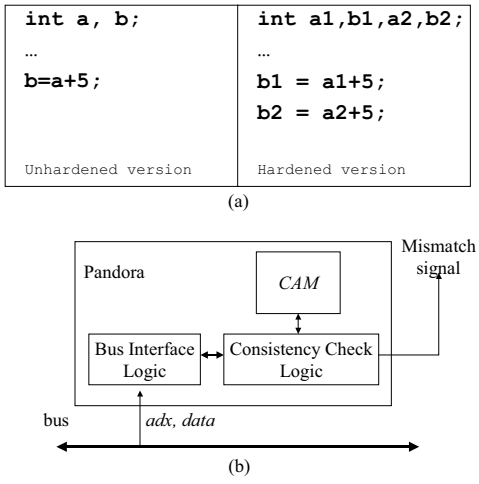


Fig. 7. The conceptual schema of the enhanced hybrid approach including data checking [8]. An example of the required software modification is reported in (a), while in (b) the I-IP structure including the CAM memory is shown; this CAM module stores the information for the comparison of the replicated data.

This integrated solution mainly consists in modifying the code by duplicating the data space and instructions, and in inserting a suitable CAM memory storing information about the replicated variables: in this enhanced solution, the consistency-checking module is also in charge of identify the variable consistency. A conceptual schema of such solution for data checking is shown in Figure 7.

Currently, we are working towards the integration of the two approaches and towards the development of a single I-IP integrating the functions of both.

We are also analyzing the effectiveness and possible modification to be introduced in the approach when it is applied to a SoC including a superscalar processor.

6. 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 Transactions on Computers, Vol. 37, No. 2, Feb. 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-critical software with respect to transient errors", IEEE Transactions on Nuclear Science, Vol. 47, No. 6, Dec. 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, Mar. 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, Feb. 2002, pp. 180-199

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

[8] L. Bolzani, M. Rebaudengo, M. Sonza Reorda, F. Vargas, M. Violante, "Hybrid Soft Error Detection by means of Infrastructure IP cores", IEEE International On-Line Testing Symposium, 2004, pp. 79-84

[9] A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques and Tools, Addison-Wesley, 1986

[10] B. Randell, "System Structure for Software Fault Tolerant," IEEE Transactions On Software Engineering, Vol. 1, No. 2, Jun. 1975, pp. 220-232

[11] A. Avizienis, "The N-Version Approach to Fault-Tolerant Software," IEEE Transactions On Software Engineering, Vol. 11, No. 12, Dec. 1985, pp. 1491-1501

[12] K. H. Huang, J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations", IEEE Transactions on Computers, vol. 33, Dec 1984, pp. 518-528

[13] S.S. Yau, F.-C. Chen, "An Approach to Concurrent Control Flow Checking", IEEE Transactions on Software Engineering, Vol. 6, No. 2, March 1980, pp. 126-137

[14] Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy, J.A. Abraham, "Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection", IEEE Transactions on Parallel and Distributed Systems, Vol. 10, No. 6, Jun. 1999, pp. 627-641

[15] N. Oh, P.P. Shirvani, E.J. McCluskey, "Control-Flow Checking by Software Signatures", IEEE Transactions on Reliability, Vol. 51, No. 2, Mar. 2002, pp. 111-122

[16] M. Namjaoo, E. J. McCluskey, "Watchdog processors and capability checking", International Symposium on Fault Tolerant Computing, 1982, pp. 245-248

[17] A. Mahmood, D.J. Lu, E. J. McCluskey, "Concurrent fault detection using a watchdog processor and assertions", IEEE International Test Conferemce, 1983, pp. 622-628

[18] N.R. Saxena, E.J. McCluskey, "Control Flow Checking Using Watchdog Assists and Extended-Precision Checksums", IEEE Transactions on Computers, Vol. 39, No. 4, Apr. 1990, pp. 554-559

[19] M. Namjoo, "CERBERUS-16: An architecture for a general purpose watchdog processor", International Symposium on Fault-Tolerant Computing, 1983, pp. 216-219

[20] J. Ohlsson, M. Rimen, "Implicit Signature Checking", Int. Symp. on Fault-Tolerant Computing, 1995, pp. 218-227

[21] K. Wilken, J.P. Shen, "Continuous signature monitoring: low-cost concurrent detection of processor control errors", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 9, No. 6, June 1990, pp. 629-641

[22] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, M. Violante, "Soft-error Detection Using Control Flow Assertions", IEEE Int. Symp. on Defect and Fault Tolerance in VLSI Systems, 2003, pp. 581-588

[23] 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

[24] F. L. Vargas, M. Nicolaidis, "SEU-Tolerant SRAM Design Based on Current Monitoring", IEEE International Symposium on Fault-Tolerant Computing, Jun. 1994, pp. 106-115

[25] 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, Jun. 2002, pp. 261-271

[26] www.eembc.org