A New Hybrid Fault Detection Technique for Systems-on-a-Chip

Paolo Bernardi, *Student Member*, *IEEE*, Leticia Maria Veiras Bolzani, *Student Member*, *IEEE*, Maurizio Rebaudengo, *Member*, *IEEE*, Matteo Sonza Reorda, *Member*, *IEEE*, Fabian Luis Vargas, *Member*, *IEEE*, and Massimo Violante, *Member*, *IEEE*

Abstract—Hardening SoCs against transient faults requires new techniques able to combine high fault detection capabilities with the usual requirements of SoC design flow, e.g., reduced design-time, low area overhead, and reduced (or null) accessibility to source core descriptions. This paper proposes a new hybrid approach which combines hardening software transformations with the introduction of an Infrastructure IP with reduced memory and performance overheads. The proposed approach targets faults affecting the memory elements storing both the code and the data, independently of their location (inside or outside the processor). Extensive experimental results, including comparisons with previous approaches, are reported, which allow practically evaluating the characteristics of the method in terms of fault detection capabilities and area, memory, and performance overheads.

Index Terms—SoC dependability, infrastructure IP, transient fault detection.

1 Introduction

It is now commonly accepted that electronic circuits are increasingly affected by transient faults and that the chance for a circuit to produce an incorrect output due to transient faults is becoming not negligible, even at ground level. The reasons for this situation are mainly related to the introduction of new semiconductor technologies whose higher integration level, smaller device size, and higher working frequency make circuits more sensitive to the effects of particles strike, as well as to other phenomena (such as crosstalk and electromagnetic interference) able to produce transient faults.

At the same time, electronic systems are being adopted in a high number of applications where dependability is a prime concern; in some of these fields of applications (such as automotive, biomedical, and telecommunication), cost and time-to-market are also very important, thus forcing designers and system engineers to develop new techniques able to guarantee high levels of dependability and suitable for easy adoption within the existing design flow, without introducing unacceptable hardware and performance overhead.

More particularly, when System on Chip (SoC) products are developed for such a kind of applications, there is often the need for effective techniques able to guarantee a given degree of dependability without radically changing the usual SoC design approach, which is based on reusing existing IP cores, possibly coming from third parties. On the other side, strict cost requirements often make unfeasible other approaches based on replacing (by adopting hardened versions) either the manufacturing technology or the component library. Therefore, when dealing with SoC systems, dependability-oriented techniques should not be based on modifying the functional cores, but they should rather act on the whole system architecture, possibly adding new cores devoted to dependability enhancement. Following this strategy, a new family of IP cores, denoted *Infrastructure IP cores* (I-IPs), has recently been introduced not only to guarantee dependability, but also to support a wide set of possible goals, such as test, silicon debug, diagnosis, etc. [1].

When considering processor-based SoCs, a possible approach to guarantee dependability lies in the adoption of the so-called Software Implemented Hardware Fault Tolerance (SIHFT) techniques [4], [8], [11]. They are based on modifying the software executed by the processor (or controller) introducing some sort of redundancy so that the occurrence of faults is detected. SIHFT techniques are characterized by their ease of use since they only require modification to the software, while the hardware is not modified, and have been proven to be able to guarantee a rather high degree of fault coverage. However, their adoption is often limited by the high overhead they introduce, both in terms of increased code size and of decreased performance.

The contribution of this paper resides in the development of a new Infrastructure IP suitable to be adopted to improve the reliability of processor-based SoCs: The goal of this I-IP is to guarantee high detection capability with respect to transient faults affecting data and code memory, as well as the memory elements within the processor (e.g., caches, user registers, hidden memory elements such as the control unit's flip flops). The proposed method combines together methods devised to deal with faults affecting the

- *P. Bernardi, L.M. Veiras Bolzani, M. Rebaudengo, M. Sonza Reorda, and M. Violante are with the Dipartimento di Automatica e Informatica, Politecnico di Torino, c.so Duca degli Abruzzi 24, 10129, Torina, Italy. E-mail: {Paolo.Bernardi, Leticia.VeirasBolzani, Maurizio.Rebaudengo, Matteo.SonzaReorda, Massimo.Violante}@polito.it.*
- *F.L. Vargas is with the Departamento de Engenharia Elétrica, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), Av. Ipiranga, 6681-CEP 90619-900, Porto Alegre-RS-Brazil. E-mail: vargas@computer.org.*

Manuscript received 6 Dec. 2004; revised 16 Mar. 2005; accepted 11 May 2005; published online 21 Dec. 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-0405-1204.

0018-9340/06/$20.00 © 2006 IEEE Published by the IEEE Computer Society

data with others dealing with faults affecting the program execution flow. As a result, this technique provides a very high degree of fault detection, independently of the location of the fault.

The proposed approach is a *hybrid* one since it combines the adoption of some SIHFT techniques in a minimal version (thus reducing their cost) with the introduction of the I-IP into the SoC: In this way, we can combine the benefits of hardware techniques (e.g., low performance overhead) with the flexibility, ease of use, and low cost of software techniques. In particular, when comparing this method with previously proposed ones (e.g., those based on watchdog processors [16]), it is important to note that the former is much more flexible and easier to adopt: The I-IP is completely independent of the application run by the processor and, thus, it does not have to be modified when any change in the application software is introduced; moreover, the functions it implements can be conveniently implemented with a negligible cost in terms of area occupation.

The new hybrid approach has been evaluated both from a theoretical point of view and from an experimental one: The proposed I-IP has been implemented and adopted in a sample SoC, including an Intel 8051 microcontroller core. A set of benchmark programs have been considered and several fault-injection campaigns have been performed to assess the fault detection capabilities of the method.

As a major conclusion, we claim that the new method has higher detection capabilities with respect to pure SIHFT techniques, while showing reduced code and performance overheads. On the other side, the I-IP cost in terms of area overhead is reduced (less than 5 percent for the considered case) and easier to introduce and maintain in the typical SoC design flow than a classical watchdog.

The approach proposed in this paper integrates the solutions presented in [24] and [25]; the approach presented here not only addresses faults affecting both data integrity and control flow check, but demonstrates that cleverly combining together the previous approaches results in higher detection capabilities with reduced overhead.

The rest of the paper is organized as follows: Section 2 summarizes the methods already proposed in the literature to detect transient faults in a processor-based SoC. Section 3 summarizes the main characteristics of the SIHFT method we used as a starting point; Section 4 introduces the new *hybrid* method. Section 5 reports an evaluation of the proposed approach when it is applied in a system based on an Intel 8051 controller. Finally, Section 6 draws some conclusions.

2 Previous Works

Error-detection techniques for software-based systems can be organized into two broad categories: software-implemented techniques, which exploit purely software detection mechanisms, and hardware-based ones, which exploit additional hardware. Such techniques focus on checking the consistency between the expected and the executed program flow, recurring to the insertion of additional code lines or by storing flow information in suitable hardware structures, respectively.

2.1 Software-Based Techniques

Common techniques relying on software modifications exploit the concepts of information, operation, and time redundancy to detect the occurrence of errors during program execution. In the past 30 years, many techniques have been developed. The oldest techniques obtain a dependable system by replicating the execution of the program and by voting among the results produced by each replica (e.g., Recovery Blocks [9] and N-Version Programming [10]). Although effective, these approaches rely on software designers for their implementations: Programmers 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 (such as [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 we intend to harden. However, some of these techniques are not general since they are dedicated to hardening a certain class of applications, only [11].

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* [13]. A basic block is a sequence of consecutive instructions in which, in the absence of faults, the control flow 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 $P = \{V, E\}$ [14] composed of a set of nodes V and a set of edges E, where $V = \{v_1, \ldots, v_i, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_i, \ldots, e_m\}$. Each node v_1 represents a basic block and each edge e_i represents the branch $br_{i,j}$ from v_i to v_j. The edge $br_{i,j}$ is not necessarily an explicit branch instruction, but it may also represent a jump, a subroutine call, or a return instruction.

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 succeeding v_i and $pred(v_i)$ as the set of nodes predeceding 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. A branch from block i to j is *illegal* if $br_{i,j}$ is not included in E. This illegal branch indicates a control-flow error, which could be caused by transient or permanent faults. Two types of control-flow errors can be defined:

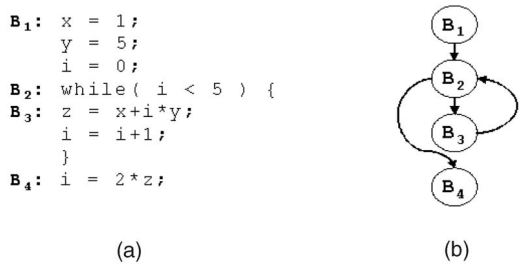

Fig. 1. Example of source code and its corresponding program graph.

- *Interblock faults* that consist of illegal branches $br_{i,j}$, where $br_{i,j} \notin E$ and $i \neq j$, i.e., branches between two different basic blocks.
- *Intrablock faults* that consist of illegal branches $br_{i,i}$, where $br_{i,i} \notin E$, i.e., branches within the same basic block.

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

Among the most important solutions proposed in the literature, there are the techniques called *Enhanced Control Flow Checking Using Assertions* (ECCA) [3] and *Control Flow Checking by Software Signatures* (CFCSS) [4]. 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. A **test** assertion is executed at the beginning of the block, which checks if the previous basic block is permissible, according to the Program Graph; a divide by zero error identifies the control flow error. A **set** assignment is executed at the end of the block and updates the identifier, taking into account the whole set of possible next basic blocks. ECCA is able to detect all the single interblock control-flow errors. It is not able to detect intrablock control-flow errors nor 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 runtime signature. In the 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 each 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, 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 [4], given a graph with the set of edges $E = \{br_{1,4}, br_{1,5}, br_{2,5}, br_{2,6}, br_{3,5}, br_{3,6}\}$, an illegal branch between node v_1 to node v_6 corresponding to the branch $br_{1,6}$ is not detected by the method. As far as faults affecting program data are concerned, several techniques have recently been proposed (for example, [2], [15]), which exploit information and operation redundancies. The most recently introduced approaches modify the source code of the application to be hardened against faults by introducing information and operation redundancies. Moreover, consistency checks are added to the modified code to perform error detection. In order to explain the rationale behind the approaches in [2] and [15], let us consider the code in Fig. 1. The approach proposed in [2] exploits several code-transformation rules that mandate for duplicating each variable and each operation among variables. Moreover, each time a variable is read, a consistency check between the variable and its replica should be performed. When applied to the considered example, the approach proposed in [2] produces the code reported in Fig. 2, where the instructions added to harden the program are reported in boldface.

```
x0 = 1; x1 = 1;
y0 = 5; y1 = 5;
i0 = 0; i1 = 0;
while( i0 < 5 ) {
if( i0 != i1 ) error();
       z0 = x0+i0*y0;
       z1 = x1+i1*y1;
       if( i0 != i1 ) error();
       if( x0 != x1 ) error();
       if( y0 != y1 ) error();
       i0 = i0+1;
       i1 = i1+1;
       if( i0 != i1 ) error();
}
i0 = 2*z0;
i1 = 2*z1;
if( z0 != z1 ) error();
```

Fig. 2. Code hardened according to [2].

Conversely, the approach proposed in [15], named *Error Detection by Data Diversity and Duplicated Instructions* (ED^{4}I), consists of developing a modified version of the program, which is executed along with the unmodified program. After executing both the original and the modified versions, their results are compared: An error is detected if any mismatch is found. An example of the modified version of the code according to this approach for the considered example is reported in Fig. 3. Both the approaches introduce some overheads. Additional memory is needed for storing the replicated variables and the additional instructions for duplication of operations and for consistency checks. Moreover, the execution time is increased by the duplicated instructions and by the consistency checks. By introducing consistency checks performed each time a variable is read, the approach proposed in [2] minimizes the latency of faults;

```
x0 = 1;
y0 = 5;
i0 = 0;
while( i0 < 5 ) {
       z0 = x0+i0*y0;
       i0 = i0+1;
}
i0 = 2*z0;
x1 = -2;
y1 = -10;
i1 = 0;
while( i1 > -10 ) {
       z1 = x1+i1*y1/(-2);
       i1 = i1+(-2);
}
i1 = 2*z1;
if( i0 != i1 ) error();
```

Fig. 3. Code hardened according to [15].

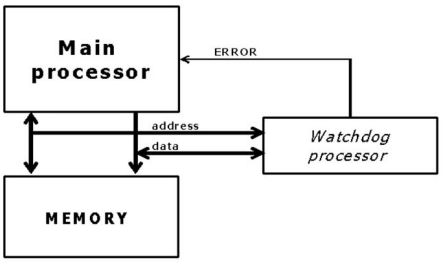

Fig. 4. The basic architecture of a system hardened through a watchdog processor.

however, it is only suitable for detecting transient faults since the same operation is repeated twice. Conversely, the approach proposed in [15] exploits diverse data and duplicated instructions and, thus, it is suitable for both transient and permanent faults. As a drawback, since a consistency check is performed only after the two replicas of the program have been executed, the fault latency is generally greater than in [2]. Moreover, the adoption of the ED^{4}I technique requires a careful analysis of the size of used variables since it may otherwise trigger overflow situations.

Software-implemented techniques are appealing since they do not require modifications to the hardware running the hardened application and, thus, they can be implemented with very low costs. However, although very effective in detecting faults affecting both programs execution-flow and programs data, the software-implemented approaches may introduce significant time overheads that limit their adoption only to those applications where performance is not a critical issue.

2.2 Hardware-Based Techniques

Hardware-based techniques exploit special-purpose hardware modules, called *watchdog processors* [16], to monitor the control-flow of programs, as well as memory accesses. The watchdog monitors the behavior of the main processor running the application code, according to the architecture of Fig. 4.

A watchdog processor can perform three types of monitoring operations. *Memory-accesses checks* consist of monitoring for unexpected memory accesses executed by the main processor. As an example, an approach is proposed in [17] 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.

Consistency check of variables contents consists of 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 main processor writes or reads through range checks or by exploiting known relationships among variables [18].

Control-flow check consists of controlling whether all the taken branches are consistent with the Program Graph of the software running on the main processor [20], [21], [22], [23]. As far as the control-flow check is considered, two types of watchdog processors may be envisioned. An *active watchdog processor* executes a program concurrently with the main processor. The Program Graph of the watchdog program is omomorphic to the main processor one. During program execution, the watchdog continuously checks whether its program evolves as the one executed by the main processor [21]. 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 nonnegligible. *Passive watchdog processors* do not execute any program; conversely, they compute a signature by observing the main processor's bus. Moreover, they perform consistency checks each time the main program enters/leaves a basic block within the Program Graph. A cost-effective implementation is described in [22], where a watchdog processor observes the instructions the main processor executes and computes a runtime 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 precalculated signature, while the main processor executes a NOP instruction. The watchdog processor compares the received precomputed signature with the one computed at runtime and it issues an error signal in case of mismatch. An alternative approach is proposed in [23], where the watchdog processor computes a runtime 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 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 Software-Based Hardening Approach

The hybrid approach presented here is built over a modified software-based technique: Its purpose is to improve the detection abilities offered by purely software approaches by investigating the trade-off between the software and the hardware overheads. To better explain the fundamentals of the proposed technique, the description of the original software-based method on the base of this work is provided.

Mainly, the adopted approach exploits the code transformation rules described in [2] and [8]. The fault model addressed by the approach is the transient single bit-flip in the memory elements. This model is used to represent *Single Event Upsets* (SEUs) in memory bits, which are likely to happen due to the decrease in the magnitude of the electric charges used to carry and store information [2]. Moreover, in this paper, transient faults affecting both the memory segments, data, and code, and the processor internal memory elements (register file and control registers) are considered.

In developing this approach (similarly to [15]), it is assumed that the application being hardened does not exploit dynamic memory allocation: All the data structures are defined statically at compile time. This is not a significant limitation for developers of embedded applications, who

```
if( test(S0,2) ) error();
set(S1,1);
x = 1;
y = 5;
i = 0;
if( test(S1,1) ) error();
set(S1,2);
while( i < 5 ) {
      if( test(S1,2) ) error();
      if( test(S3,2) ) error();
      set(S3,1);
      z = x+i*y;
      i = i+1;
      if( test(S3,1) ) error();
      set(S3,2);
}
if( test(S1,2) ) error();
if( test(S3,2) ) error();
set(S4,1);
i = 2*z;
if( test(S4,1) ) error();
set(S4,2);
```

Fig. 5. Code hardened according to the rules for code checking.

sometimes are forced to code standards that already avoid dynamic memory usage.

3.1 Code Checking

To detect faults affecting the code, we refer to the approach presented in [8]. The method is based on a set of rules applied to the high-level code. The goal of the transformations is to detect those faults modifying the program control flow in such a way that incorrect jumps are executed (either by transforming functional instructions into jump instructions or by modifying existing jump instructions or their operands). A dedicated global integer variable (called *code*) is introduced for that purpose: It contains a signature updated at runtime to check whether the program correctly traverses the Program Graph. Two signature values, $S_{i,1}$ and $S_{i,2}$, defined at compile time, are associated to basic block B_i: These two values are written in the code variable when the block is entered and left, respectively.

The following functions[1] are introduced in order to implement the mentioned hardening technique. A **test** function controls the current value of the code variable and checks if it is permissible, according to the Program Graph. A **set** function updates the code variable by setting it to a new value. Calls to the test and set functions are introduced at the beginning and at the end of each block. This approach allows covering all of the single intra and interblock control flow faults, as demonstrated in [8].

The hardened version of the code reported in Fig. 1, generated applying the above transformation rules, is reported in Fig. 5. An $error()$ function is executed when a wrong behavior is detected.

When the above rules are combined with those for data checking (described below), further faults affecting the program execution flow can be detected.

3.2 Data Checking

Data hardening is performed according to the following rules [2]:

1. From an implementation point of view, the two functions may indeed correspond to macros, thus minimizing their cost in terms of execution time.

```
if( test(S0,2) ) error();
set(S1,1);
x0 = 1; x1 = 1;
y0 = 5; y1 = 5;
i0 = 0; i1 = 0;
if( (x0!=x1) || (y0!=y1) || (i0!=i1) )
      error();
if( test(S1,1) ) error();
set(S1,2);
while( i < 5 ) {
      if( test(S1,2) || test(S3,2) )
            error();
      set(S3,1);
      z0 = x0+i0*y0;
      z1 = x1+i1*y1;
      i0 = i0+1;
      i1 = i1+1;
      if( (x0!=x1) || (y0!=y1) ||
          (i0!=i1) || (z0!=z1) )
            error();
      if( test(S3,1) ) error();
      set(S3,2);
}
if( test(S1,2) || test(S3,2) )
      error();
set(S4,1);
i0 = 2*z0; i1 = 2*z1;
if( test(S4,1) ) error();
set(S4,2);
```

Fig. 6. Code hardened by applying transformation rules for combined data and code checking.

- Every variable x must be duplicated: Let x_0 and x_1 be the names of the two copies. Two sets of variables are thus obtained, the former ($set(v_0)$) holding all the variables with footer 0 and the latter ($set(v_1)$) holding all the variables with footer 1.
- Every write operation performed on a variable x must be performed on its replica x_0 in v_0 (using only variables belonging to v_0) and its replica x_1 in v_1 (using only variables belonging to v_1); after each read operation on a variable x, the two replicas x_0 and x_1 must be checked and, if an inconsistency is detected, an error procedure is activated.

3.3 Combined Data and Code Checking

The combined application of the presented hardening rules allows the generaton of a redundant version of a safety-critical program able to detect faults affecting both the code and the data memory areas. The hardened version of the code reported in Fig. 1 generated by applying all the transformation rules for data and code checking is reported in Fig. 6.

4 The Hybrid Approach

In order to overcome the limits of the purely software approach presented in Section 2.1, we devised a hybrid solution tailored to be applied in SoC devices. The main idea is to still retain the approach described in Section 3, but to resort to an external Infrastructure IP to reduce its cost and enhance its performance in terms of fault detection capabilities.

The result is a *hybrid* approach where fault detection-oriented features are still implemented in software, but most of the computational efforts are demanded to external hardware. The executed program allows the processor to communicate with an external circuitry through the SoC bus: By computing the received information, this circuitry determines incorrect executions.

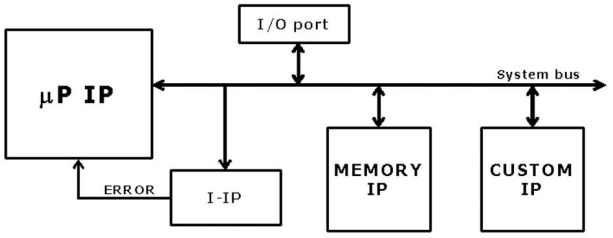

Fig. 7. Architecture of the generic SoC system including the fault detection-oriented I-IP.

The goal is to devise a method that can be easily adopted in the typical SoC design flow, i.e., it requires minimal changes in the hardware (apart from the insertion of the I-IP), while the software is simplified with respect to the purely software fault detection approach overviewed in Section 3. A further constraint is the flexibility of the approach: Any change in the application should result in software changes only, while the hardware (including the I-IP) should not be affected. This means that the I-IP must not include any information about the application code, but must be general enough to be able to protect any code, provided that it has been hardened according to the suggested approach.

The I-IP we propose is connected to the system bus as an I/O peripheral interface. This means that the I-IP can observe all the operations performed on the bus by the processor and can be the target for some write operations performed by the processor at specific addresses of the memory or I/O address space (depending on the adopted I/O scheme). When the I-IP detects an error, it activates an ERROR signal, which can be sent either to the processor or to the outside, depending on the preferred recovery scheme. The architecture of the system including the I-IP is reported in Fig. 7.

Our method can be introduced more easily by first considering, in a separate manner, the techniques adopted for dealing with faults affecting the code and those dealing with faults affecting the data. However, the two sets of techniques are supported in an integrated manner by the I-IP, resulting in even higher fault detection capabilities with respect to the purely software approach.

4.1 Support for Control Flow Checking

The basic idea behind the proposed hybrid approach for checking the correct control-flow execution is that we can simplify the hardened code and improve its performance by moving, in hardware, the control flow checks. According to our solution, the code is in charge of signaling the I-IP when a new basic block is entered. Since the I-IP is not intended to record any information about the application code, the hardened program must send to the I-IP all the information required to check whether the new block can be legally entered given the list of previous blocks. The I-IP records, in an internal register, the current signature. Once it is informed that a new block is entered and it has received the list of blocks that can legally reach the new block, it checks whether the stored signature is included in this list. If not, the ERROR signal is raised. Otherwise, the current signature is updated with the signature of the new block.

```
IIPtest(S0,2);
IIPset(S1,1);
x = 1;
y = 5;
i = 0;
IIPtest(S1,1);
IIPset(S1,2);
while( i < 5 ) {
      IIPtest(S1,2);
      IIPtest(S3,2);
      IIPset(S3,1);
      z = x+i*y;
      i = i+1;
      IIPtest(S3,1);
      IIPset(S3,2);
}
IIPtest(S1,2);
IIPtest(S3,2);
IIPset(S4,1);
i = 2*z;
IIPtest(S4,1);
IIPset(S4,2);
```

Fig. 8. Control-flow check according to the hybrid approach.

In order to support communication between the processor and the I-IP, we introduced two high-level functions, $IIPtest()$ and $IIPset()$, whose role is the following:

- $IIPset(B_i)$ informs the I-IP that the program has just entered into basic block B_i.
- $IIPtest(B_j)$ informs the I-IP that block B_j belongs to the set of the predecessors of the newly entered block.

The I-IP contains two registers, A and B, that can be accessed by the processor by performing a write operation at a couple of given addresses: $IIPtest(B_j)$ informs the I-IP that block B_j belongs to the set of the predecessors of the newly entered block X_A and X_B. The two functions $IIPtest()$ and $IIPset()$ are translated into write operations at the addresses X_A and X_B, respectively, thus resulting in a very limited cost in terms of execution time and code size. The parameter of the function is written in the register, thus becoming available to the I-IP for processing. A sequence of calls to the two functions should be inserted in the code at the beginning and at the end of each block B_k. First, a call to $IIPtest(B_i)$ is inserted for any block $B_i \in prev(B_k)$. Then, a call to $IIPset(B_k)$ is inserted. When noticing a write operation on register A, the I-IP set or reset an internal flag, depending on the result of the comparison between the function parameter and the internally stored signature. When noticing a write operation on register B, the I-IP verifies the value of the flag and possibly activates the ERROR signal. Otherwise, the signature of the current block is updated using the value written in register B.

A code portion for the same example introduced in Fig. 1 that adopts the proposed approach, i.e., sending information to the I-IP, is reported in Fig. 8. Two functional parts can be distinguished in the I-IP to execute concurrent control-flow checking: `Bus Interface Logic` and `Control Flow Consistency Check Logic`. Such a schematic circuitry subdivision is highlighted in Fig. 11.

The `Bus Interface Logic` implements the interface needed for communicating with the processor bus.

The `Control Flow Consistency Check Logic` is in charge of verifying whether any control flow error affects the application expected behavior and informing the system

```
IIPtest(S0.2);
IIPset(S1.1);
x0 = 1; x1 = 1;
y0 = 5; y1 = 5;
i0 = 0; i1 = 0;
IIPtest(S1.1);
IIPset(S1.2);
while( i0 < 5 ) {
      IIPtest(S1.2);
      IIPtest(S3.2);
      IIPset(S3.1);
      z0 = x0+i0*y0;
      z1 = x1+i1*y1;
      i0 = i0+1;
      i1 = i1+1;
      IIPtest(S3.1);
      IIPset(S3.2);
}
IIPtest(S1.2);
IIPtest(S3.2);
IIPset(S4.1);
i0 = 2*z0; i1 = 2*z1;
IIPtest(S4.1);
IIPset(S4.2);
```

Fig. 9. The full implementation of the hybrid approach.

through the error signal if error detection happened. The I-IP is internally provided with the circuitry to both store and update the current signature each time data are sent from the processor: Such circuitry calculates, at runtime, the value of the masks, as described in Section 3.

4.2 Support for Data Checking

When considering the faults affecting the data, our approach is based on the idea of moving, in hardware, the task of comparing the two replicas of a variable each time it is accessed. In this way, the hardened code is significantly simplified: Not only is its size reduced and the performance increased, but a number of conditional jump instructions are removed, thus reducing the risk for additional faults affecting the code.

To implement the above idea, the I-IP must monitor the bus, looking for memory read cycles. In principle, the I-IP should simply identify the two cycles accessing the two replicas of the same variable, checking whether their value is identical. If not, an error is detected.

To implement this idea, a mechanism is required allowing the I-IP to know the addresses of the two replicas of the same original variable and to understand whether a given address corresponds to the first or second replica. A solution to this issue will be further described in this section.

Moreover, it is important to note that the two bus cycles accessing the two replicas of the same variable are not necessarily consecutive. In fact, the compiler often reorganizes the assembly code so that instructions are reordered in such a way that the two instructions are interleaved with others. However, in developing our I-IP we assumed that the compiler never modifies the code in such a way that the second replica of a variable is accessed before the first replica. To tackle this issue, the I-IP contains a CAM memory, which is used to store the address-data couple corresponding to each variable accessed in memory, whose replica has not been accessed, yet. The CAM is indexed with the address field. In more detail, the I-IP implements the following algorithm:

- If a memory read is detected on the bus, the address and data values are captured.

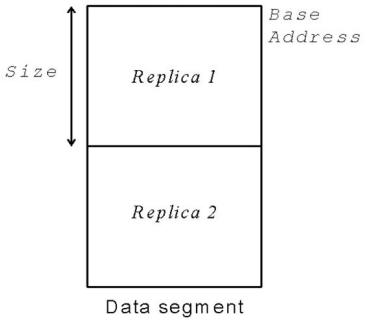

Fig. 10. Data segment of the hardened program.

- If the read operation relates to the first replica of a variable, a new entry is inserted in the CAM, containing the just captured address and data values.
- If the read operation relates to the second replica of a variable, an access is made to the CAM:
  - If an entry with the same address is not found, the $ERROR$ signal is raised.
  - Otherwise, the data is compared with the one stored in the CAM entry and the ERROR signal is raised in the case of a mismatch.
  - The entry is removed from the CAM.

This algorithm has several interesting properties. It detects all the faults affecting the data that can be detected by the purely software approach. It can be straightforwardly (and inexpensively) extended to deal with write operations, too. A separate CAM is reserved for entries related to write operations. Thanks to this extension, some faults that cannot be detected by the purely software approach are detected by the hybrid one. When the end of a basic block is reached, the CAM should be empty since the two replicas of all the variables should have been accessed. If this is not the case, an error (likely, a control flow error) has happened: The $ERROR$ signal is raised.

An example of how the hardened code of the example should be modified according to the above approach is reported in Fig. 9.

As we mentioned before, an efficient mechanism is required to allow the I-IP to distinguish whether a given address identifies the first or second replica of a variable and to compute the address of the first replica once that of the second one is available. The current solution we adopted assumes that the data segment of the program is divided into two portions, as shown in Fig. 10. The upper portion contains the first replica of each variable, while the lower one stores the second replica. This solution can be easily implemented acting on the options of C compilers.

The above assumption about variable location in memory eases the task of dealing with the two replicas of the same variable. In more detail, as soon as a memory access cycle is detected on the bus, the two fields (adx, $data$) are extracted, corresponding to the address and value of the accessed variable, respectively. With *Base* begin the beginning address of the data segment and *Size* being the size of each portion of the segment, if $adx < Base + Size$, then the first replica of the variable is currently being accessed;

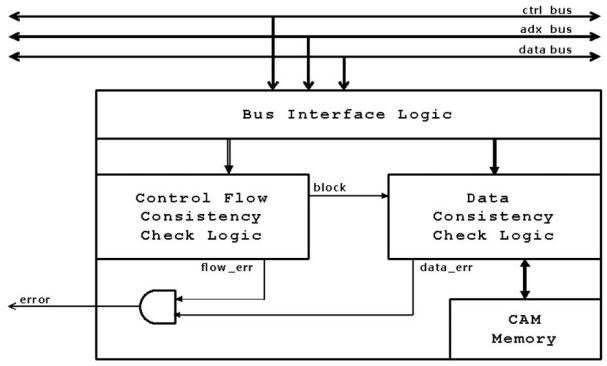

Fig. 11. Schematic architecture of the I-IP implementing control flow and data checking.

otherwise, the second replica is being accessed. To compute the address of the first replica when the address $adx2$ of the second is available, the following expression is used:

$$adx1 = adx2 - Size. \quad (1)$$

Three functional parts can be distinguished in the I-IP circuitry devoted to executing concurrent data checking, as reported in Fig. 11: These parts are named `Bus Interface Logic`, `Data Consistency Check Logic`, and `CAM Memory`. The `Bus Interface Logic` is shared with the circuitry devoted to control flow checking and implements the interface for accessing to the processor bus. It is able to decode 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 addresses and values are then forwarded to the `Data Consistency Check Logic`.

The `Data Consistency Check Logic` implements the consistency checks verifying whether any data stored in the memory or the processor has been modified. For this purpose, as soon as a new couple (adx, $data$) is extracted from the bus, it computes the address of the corresponding replica, accesses the `CAM memory`, and verifies whether the searched entry exists. In the positive case, it compares *data* with the data field of the entry (possibly raising the error signal in case of mismatch) and then removes the entry from the `CAM`. In the negative case, it inserts a new entry in the `CAM`.

Moreover, considering the program structure presented at the beginning of this paragraph, each instruction into a basic block has a replica within the same block. Consequently, we can assume that the `CAM memory` is empty when a new basic block is entered. To cope with this assumption, the `Data Consistency Check Logic` receives a block signal, generated by the `Control Flow Consistency Check Logic`. Such a signal is asserted when exiting a basic block: The content of the `CAM Memory` is then checked and, if not empty, the $error$ signal sets on.

The proposed I-IP design, whose schematic is shown in Fig. 11, can be easily adapted to different processors: Both the `CAM Memory` and `Consistency Check Logic` modules are parametric and can thus be reused for different address and data sizes. Only the `Bus Interface Logic` needs to be reworked for adapting to the bus protocol implemented by different processors. When the I-IP is introduced in an SoC, the only customization required concerns the addresses X_A and X_B of the two registers written by the $IIPtest()$ and $IIPset()$ procedures, respectively, and the values of $Base$ and $Size$, as graphically illustrated in Fig. 10.

4.3 Error Detection Capabilities

To experimentally assess the effectiveness of our hybrid approach, we first developed a categorization of the possible faults affecting the memory elements of a system and then theoretically analyzed the error detection capabilities of the proposed method. Faults can be divided into the following types, according to the module affected by the considered bit flip: *memory code area*, *memory data area*, and *processor internal memory elements*.

When considering the first fault type, we considered the processor instruction set as divided into two instructions categories: *functional* instructions (executing some sort of processing on data, such as transfer, arithmetic operations, bit manipulation, etc.) and *branch* instructions. Consequently, whereas the modified bit in the code area belongs to an opcode, the following categories can be introduced:

- `functional_to_branch`: The modified bit in the opcode transforms a functional instruction into a branch instruction.
- `branch_to_functional`: The modified bit in the opcode transforms a branch instruction into a functional instruction.
- `functional_to_functional`: The opcode of a functional instruction is transformed into another functional instruction:
  - with the same number of operands.
  - with a different number of operands.
- `branch_to_branch`: The opcode of a branch instruction is transformed into another branch instruction:
  - with the same number of operands.
  - with a different number of operands.

In the case of a `functional_to_branch` code modification, the program flow is guaranteed to change; if the target of the branch introduced by the fault is out of the basic block boundary, both software and hybrid detection mechanisms detect the fault. On the contrary, when the branch target is inside the currently executed basic block, software detection may fail, while hybrid successfully copes with most such faulty behaviors, thanks to the additional check on the `CAM` memory emptiness performed at the end of each block. These faults could also lead to a timeout if the target of the faulty jump is a previous instruction within the same basic block.

Faults belonging to the `branch_to_functional` category also cause a change in the program control flow. If the new instruction has the same number of operands as the original, the detection is guaranteed by both approaches thanks to the consistency check (for the software approach) and to the data checking techniques (for the hybrid approach). On the other hand, if the new instruction has a different number of operands, in the software approach, the

fault may not be detected because the consistency checks can only evaluate the equivalence between two variables and are not able to evaluate possible misalignments into the code, while the hybrid approach is able to detect such types of faults thanks to its capability to store all the memory accesses and verify possible unbalanced memory accesses.

For `functional_to_functional` code modifications, if the number of required operands of the exchanged instruction is the same in the original and faulty instructions, both approaches are able to detect the fault. Unfortunately, if the number of operands is changed, neither the software nor the hybrid approach can always guarantee the detection: In this case, it is possible that the modified program execution continues until the end, producing a wrong answer, even if the probability of this situation is really low. In fact, the program usually backs to its normal flow, with unexpected `CAM` memory content.

For `branch_to_branch` code modifications, in the case of unchanged number of operands, it is possible that the modified program execution continues until the end, producing a wrong answer due to incorrect condition evaluation or, more frequently, an endless loop finally resulting in the time-out condition is entered. If the number of required operands is modified, it can happen that the end of the program is reached and a wrong answer produced. As for the `functional_to_functional` code modification, the probability of such an event is low as the program usually returns to its normal flow, with unexpected `CAM` memory content. In fact, the program usually returns to its normal flow, with unexpected `CAM` memory content.

If the faulty bit corresponds to the operand of an instruction, the following unexpected program behaviors have to be investigated:

- `wrong_memory_access`: The modified operand is the address of a variable.
- `wrong_immediate_value`: The modified operand is an immediate value.
- `wrong_branch_offset`: The modified operand is the target of a branch instruction.

Considering the faults belonging to the `wrong_memory_access` category, they are covered by all the approaches, as they modify only one of the two replicas; therefore, the fault is detected by the data checking techniques.

The `wrong_immediate_value` fault category is covered by both the software and hybrid approaches. We consider the following cases: If the involved instruction is a comparison executed immediately before a branch instruction, the fault is covered by the data checking techniques at the beginning of the basic block; otherwise, the fault effect modifies the value of one replica of the variables and it is detected by the data checking techniques.

Finally, when a wrong address is accessed after branch, that is, the `wrong_branch_offset` code modification, a wrong answer is never produced for both the techniques analyzed; however, it is possible that a such modification leads to the timeout condition.

Considering the faults affecting the data area, we classify the effects of the faults as follows:

- `wrong_elaboration`: The value read from the memory is wrong.
- `wrong_branch_condition`: A branch condition is executed on a modified (and thus incorrect) value.

Both the software and the hybrid strategy guarantee the detection of a `wrong_elaboration` fault affecting the system thanks to the data checking techniques, while, if a `wrong_branch_condition` fault occurs, we can distinguish between two situations: The variable is never again accessed during the program, so we have a wrong answer; the variable is accessed again and a mismatch with its replica is observed. To avoid the former case, a read operation of the second replica of the variable is inserted exactly at the beginning of the basic block following the branch.

Concerning the effects of a single fault affecting the content of the processor registers, we highlighted the following cases:

- `wrong_general_purpose_value`: A general purpose register stores a wrong value.
- `wrong_configuration_value`: The processor is configured incorrectly.

Faults in the `wrong_general_purpose_value` category are usually detected by both the software and hybrid approaches thanks to the data checking techniques; however, sometimes two transfer instructions can read the value of the same register, then copy it into two replicas. Such a situation is usually generated when code optimization is used by the compiler. Finally, the impact of `wrong_configuration_value` faults on the program execution depends on the processor configuration and usually results in a wrong answer or, more easily, in a timeout condition with both approaches.

5 Experimental Results

To assess the effectiveness of our hybrid approach, we developed a prototypical implementation of the I-IP and exploited it for hardening an SoC including an Intel 8051 controller. For this purpose, the Infrastructure IP proposed in Section 4 has been described in the VHDL language and connected with a soft-core implementing the Intel 8051. Some benchmark programs were used to assess the properties of the hybrid approach in terms of detection capabilities and cost (memory overhead, performance slowdown, silicon area required by the I-IP).

To model the effects of SEUs, we exploited the *transient single bit flip* fault model, which consists of the modification of the content of a single storage cell during program execution.

The fault detection ability of the proposed hybrid approach has been separately investigated, considering SEUs modifying the content of the code memory area, SEUs affecting the data memory area, and SEUs affecting the microcontroller memory elements.

The adopted fault injection tool provides the required accuracy since it allows accessing all the memory elements the processor embeds, with a suitable time resolution [6].

In setting-up the fault injection experiments, a crucial factor is the selection of the fault list. Since the total number of possible faults is very high, we resorted to fault sampling

TABLE 1
Fault Injection Results Concerning Faults Affecting the Memory Area Storing the Code

Program	Version	Silent		Time Out		Detected		Wrong Answer	
		[#]	[%]	[#]	[%]	[#]	[%]	[#]	[%]
MTX	*Plain*	20,607	68.69	3,864	12.88	0	0.00	5,529	18.43
	Software	18,798	62.66	2,121	7.07	8,208	27.36	873	2.91
	ABFT	19,356	64.52	3,232	10.77	6,262	20.87	1,150	3.83
	ED^{4}I	19,649	65.50	4,258	14.19	5,999	20.00	94	0.31
	Hybrid	15,567	51.89	3,225	10.75	11,202	37.34	3	0.01
ELPF	*Plain*	16,071	53.57	3,339	11.13	0	0.00	10,590	35.30
	Software	18,015	60.05	5,583	18.61	5,115	17.05	1,287	4.29
	ED^{4}I	13,596	45.32	3,283	10.94	13,069	43.56	52	0.17
	Hybrid	14,448	48.16	2,751	9.17	12,750	42.50	51	0.17
LZW	*Plain*	6,852	22.84	5,469	18.23	0	0.00	17,679	58.93
	Software	10,260	34.20	9,405	31.35	9,420	31.40	915	3.05
	ED^{4}I	21,890	72.97	2,922	9.74	4,991	16.64	197	0.66
	Hybrid	8,703	29.01	7,836	26.12	13,377	44.59	84	0.28
V	*Plain*	8,178	27.26	6,093	20.31	0	0.00	15,729	52.43
	Software	10,884	36.28	10,302	34.34	7,518	25.06	1,296	4.32
	ED^{4}I	*NA*							
	Hybrid	10,743	35.81	5,136	17.12	13,734	45.78	387	1.29

for selecting an acceptable number of faults to be injected in the code and data segments and in the processor registers. For each fault injection, 30,000 single bit-flips were injected into each benchmark version. To verify the meaningfulness of the chosen number of faults, we performed several experiments selecting several sets of faults and then comparing the obtained results: We considered as appropriate a number of faults such that the different fault injection experiments returned nearly identical fault classifications. Results of each fault injection campaign are shown in Table 1, Table 2, and Table 3, which report the average of the results obtained in the different experiments.

Based on the aforementioned procedure, experiments have been performed considering four benchmark programs that are inspired by those in the EEMBC automotive/industrial suite [7]:

1. `5x5 Matrix Multiplication` (`MTX`): It computes the product of two 5x5 integer matrices.
2. `Fifth Order Elliptical Wave Filter` (`ELPF`): It implements an elliptic filter over a set of six samples.
3. `Lempel-Ziv-Welch Data Compression Algorithm` (`LZW`): It compresses data by replacing strings of characters with single codes.
4. `Viterbi Algorithm` (`V`): It implements the Viterbi Algorithm encoding for a 4-byte message.

For each such benchmark, up to five different implementations have been compared:

Plain: The plain version of the considered benchmark; no hardware or software fault detection techniques are exploited.

Software: The hardened version of the benchmark, obtained using the purely software hardened version combining the approaches described in 2 and 8.

ED^{4}I: The hardened version of the benchmark, obtained using the purely software hardening approach described in [15].

ABFT: The hardened version of the `MTX` benchmark, obtained using the purely software hardening approach described in [19].

Hybrid: The hardened version of the benchmark, obtained using the new approach proposed in this paper.

When performing the fault injection experiments, we classified fault effects as follows:

- *Silent*: The fault did not modify the execution of the program.
- *Time-out*: The fault modified the program execution in such a way that the processor entered an endless loop, thus not providing final results.
- *Detected*: The fault modified the program execution, but some of the detection features detected it.
- *Wrong Answer*: The fault modified the program execution and its output results differed from the expected ones. These faults are the most critical ones and their number should be minimized by the adopted fault detection mechanism.

5.1 Analysis of Fault Detection Capabilities

The following subsections report the experimental results gathered with several fault injection campaigns based on the environment described in [6].

5.1.1 Faults Affecting the Code

Results gathered when injecting 30,000 randomly selected single bit flips in the memory area storing the code of each benchmark program (in the five considered versions)

TABLE 2
Fault Injection Results Concerning Faults Affecting the Memory Area Storing the Data

Program	Version	Silent		Time Out		Detected		Wrong Answer	
		[#]	[%]	[#]	[%]	[#]	[%]	[#]	[%]
MTX	*Plain*	26,808	89.36	63	0.21	0	0.00	3,129	10.43
	Software	24,930	83.10	57	0.19	5,013	16.71	0	0.00
	ABFT	25,642	85.47	32	0.11	3,818	12.73	508	1.69
	ED^{4}I	27,029	90.10	48	0.10	2,814	9.38	109	0.36
	Hybrid	25,053	83.51	0	0.00	4,947	16.49	0	0.00
ELPF	*Plain*	28,623	95.41	33	0.11	0	0.00	1,344	4.48
	Software	22,764	75.88	30	0.10	7,206	24.02	0	0.00
	ED^{4}I	27,399	91.33	31	0.10	2,503	8.34	67	0.22
	Hybrid	24,339	81.13	0	0.00	5,661	18.87	0	0.00
LZW	*Plain*	23,889	79.63	450	1.50	0	0.00	5,661	18.87
	Software	18,642	62.14	273	0.91	11,850	36.95	0	0.00
	ED^{4}I	28,053	93.51	183	0.61	1,482	4.94	282	0.94
	Hybrid	17,958	59.86	0	0.00	12,042	40.14	0	0.00
V	*Plain*	19,137	63.79	810	2.70	0	0.00	10,053	33.51
	Software	17,067	56.89	450	1.50	12,483	41.97	0	0.00
	ED^{4}I	*NA*							
	Hybrid	17,433	58.11	0	0.00	12,567	41.90	0	0.00

TABLE 3
Fault Injection Results Concerning Faults Affecting the Processor Memory Elements

Program	Version	Silent		Time Out		Detected		Wrong Answer	
		[#]	[%]	[#]	[%]	[#]	[%]	[#]	[%]
MTX	*Plain*	27,777	92.59	927	3.09	0	0.00	1,296	4.32
	Software	27,987	93.29	189	0.63	1,734	5.78	90	0.30
	ABFT	13,968	93.13	66	0.44	829	5.53	137	0.91
	ED^{4}I	13,903	92.69	334	2.23	660	4.40	103	0.96
	Hybrid	27,039	90.13	651	2.17	2,265	7.55	45	0.15
ELPF	*Plain*	27,789	92.63	948	3.16	0	0.00	1,263	4.21
	Software	28,035	93.45	93	0.31	1,641	5.47	231	0.77
	ED^{4}I	13,618	90.79	18	0.12	1,328	8.85	36	0.24
	Hybrid	26,817	89.39	807	2.69	2,307	7.69	69	0.23
LZW	*Plain*	26,763	89.21	705	2.35	0	0.00	2,532	8.44
	Software	26,871	89.57	353	1.17	2,623	8.74	153	0.51
	ED^{4}I	14,041	93.61	102	0.68	717	4.78	140	0.93
	Hybrid	27,300	91.10	600	2.00	1,920	6.40	90	0.30
V	*Plain*	27,396	91.32	1,437	4.79	0	0.00	1,167	3.89
	Software	27,618	92.06	933	3.11	1,236	4.12	213	0.71
	ED^{4}I	*NA*							
	Hybrid	26,907	89.69	939	3.13	2,067	6.89	87	0.29

are reported in Table 1. When analyzing the reported results about injection into the code segment, the following observations can be made, which relate to the fault classification introduced in Section 4.3: First of all, the reader can easily observe that the software and ED^{4}I approaches are able to significantly reduce the number of faults leading to a wrong answer: The hybrid approach is always able to further (and significantly) decrease this number. Bit flips affecting the instruction opcode and provoking a wrong answer mainly belong to either the `functional_to_functional` category (mostly those faults alter the number of requested operands) or `branch_to_functional modifications`; the hybrid approach shows a higher detection capability with respect to these fault categories than the purely software one, mainly thanks to the check performed at the end of each basic block on the emptiness of the `CAM`. Such faults may also provoke endless program execution, falling into the timeout case. A detailed analysis of the results summarized in Table 1 showed that bit flips affecting the operands of an instruction rarely produce a wrong answer effect, as already stated in Section 4.1: Both the software and hybrid

TABLE 4
I-IP Synthesis Results Summary

Logic component	Equivalent gates [#]
Bus interface	251
Control Flow Consistency Check	741
Data Consistency Check	1,348
CAM Memory	1,736
TOTAL	4,076

approaches are able to detect this kind of fault. Additionally, purely software approaches may introduce additional branches to the Program Graph to continuously check the value of the *ERROR* flag. Moreover, the C compiler may translate some of the C instructions implementing consistency checks as sequences of assembly-level instructions containing new branches. The new branches are not protected with the test and set functions and, thus, some faults may escape software detection techniques. Conversely, when exploiting the hybrid approach, no additional branches are introduced, resulting in a lower number of faults leading to wrong answer and time out situations.

The comparison with the ED^{4}I version for `Viterbi` is not reported in Table 1, Table 2, and Table 3. The `Viterbi` program is mainly based on executing logic operations, but the authors of [15] did not explain how to apply ED^{4}I to such operations (the paper describes how to apply ED^{4}I to arithmetic operations, only). Regarding ABFT, the results included in Table 4 only refer to the `MTX` program as it is the only benchmark (among the considered ones) to which this technique can be applied. The coverage obtained by this technique to detect transient faults affecting the code segment is rather low.

5.1.2 Faults Affecting the Data

Table 2 reports the results gathered when injecting 30,000 randomly selected single bit flips in the memory area storing the data of each benchmark program. These faults are generally very likely not to produce any wrong answer situation when the software approach is adopted; the same happens with the hybrid one. The latter approach performs better than the former when faults producing a time out are considered: This is mainly due to the different behavior with respect to faults belonging to the `wrong_branch_condition` category.

The ABFT technique fails to detect some faults affecting the data segment; these escaping faults mainly belong to the `wrong_branch_condition` category.

5.1.3 Faults Affecting the Processor Memory Elements

According to the effects they produce, faults in the memory elements within the processor belong either to the `wrong_general_purpose_value` or `wrong_configuration_value` categories. The resulting behavior is very different, although both the software and the hybrid approach show low wrong answer figures, as reported in Table 3.

Complete coverage of transient faults affecting the processor memory elements can be reached by using triplication techniques (such as `TMR`), although this solution is generally undesirable because of the performance reduction and hardly applicable when the RT-level description of the processor is not available.

5.2 Overhead Analysis

The hybrid approach proposed encompasses three types of overheads with respect to the unhardened version:

- *Area* overhead, related to the adoption of an I-IP.
- *Memory* overhead, due to the insertion in the code of the *IIPtest()* and *IIPset()* functions and to the duplication of variables.
- *Performance* overhead, as additional instructions are executed.

In order to quantify the area occupation of the proposed I-IP, we designed it resorting to the `VHDL` language; the resulting code amounts to about 450 lines. The I-IP has then been synthesized using a commercial tool (*Synopsys Design Analyzer*) and a generic library developed at our institutions. The I-IP was configured to interact with the system bus of the Intel 8051 controller we adopted and it was configured with a `CAM` memory with 16 entries. The details of the resulting gate-level implementation are shown in Table 4.

When considering the overall hardened system whose size is the sum of the contributions of the Intel 8051 microcontroller and the related memories, the area overhead introduced by the I-IP is less than 5 percent. This percent area overhead is expected to further decrease when increasing the complexity of the processor, contrary to the cost for the triplication of processor memory elements, which requires, for the analyzed case study, something more than 6 percent of additional equivalent gates.

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 compared them with those of the same programs hardened according to the software-based techniques introduced in [2] and [8]. As a reference, we also measured the area occupation and program execution time of the original programs. In Table 5, we reported the observed figures. Memory occupation was measured in terms of number of bytes in the data and code segments, while duration was measured in terms of number of clock cycles for program execution.

Results reported in Table 5 show that the performance overhead of the hybrid version is, on average, about one half of that of the one with the purely software version.

When considering the memory overhead, we can observe that the increase in the size of the memory required for data is similar in the software and hybrid versions. Conversely, the memory required for the code in the hybrid version is, on average, about one half of that required by the software version.

The case of the `ELPF` program deserves special attention: This program includes several instructions writing a constant value into a variable. In the software hardened version, this translates into two variables to be written with the same value: The compiler implements this by first loading the value in a register and then copying the register

TABLE 5
Memory and Performance Overheads Summary

Program	Version	Execution time (CC)		Code size (B)		Data size (D)	
		[#]	[%]	[#]	[%]	[#]	[%]
MTX	*Plain*	13,055	-	329	-	16	-
	Software	42,584	226.19	1,315	299.70	34	112.50
	ABFT	49,792	178.27	768	233.43	32	100.00
	ED^{4}I	24,717	189.33	524	59.27	30	87.50
	Hybrid	27,930	113.94	683	107.60	34	112.50
ELPF	*Plain*	12,349	-	384	-	48	-
	Software	46,545	276.91	1,527	297.66	100	108.33
	ED^{4}I	23,136	187.35	663	72.66	62	29.17
	Hybrid	21,946	77.71	645	67.97	100	108.33
LZW	*Plain*	19,209	-	232	-	35	-
	Software	92,003	378.96	1,898	718.10	72	105.71
	ED^{4}I	35,393	184.25	878	378.45	64	82.86
	Hybrid	38,878	102.39	859	270.26	72	105.71
V	*Plain*	286,364	-	436	-	85	-
	Software	1,398,423	388.34	2,032	366.06	172	102.35
	ED^{4}I			*NA*			
	Hybrid	598,410	208.97	1,323	203.44	172	102.35

content into the variables corresponding to the two replicas of the variable. This results in less than duplicating both the code size and the program execution time.

The average block size of the two programs `LZW` and `V` is smaller than in the two other programs: This results in a proportionally higher number of $IIPtest()$ and $IIPset()$ functions inserted in the code during the hardening phase. For these reasons, `LZW` and `V` show a higher code overhead figure.

For the same reasons, the ratio between branch and functional instructions is higher in `LZW` and `V`: Since only the latter instructions are duplicated in the software and hybrid versions, this results in a relatively low performance overhead for these two programs.

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

6 Conclusions

A *hybrid* approach is presented in this paper, exploiting software-based techniques in cooperation with an I-IP core to guarantee the detection of transient faults in processor-based SoCs. The proposed approach has been evaluated by implementing the proposed I-IP and by considering its area cost in a processor-based SoC. The fault detection capabilities of our approach have been experimentally measured by performing extensive fault-injection campaigns considering realistic faults affecting memory elements.

The main advantages of the new approach are the following:

- It is suitable to be adopted in the SoC design flow since it does not require any change in the processor and memory cores, but only the insertion of a proper I-IP on the processor bus.
- The I-IP is completely independent of the code executed by the processor; when changes are introduced in this code, the I-IP stays unchanged.
- The method effectively addresses a large set of transient faults, which can be located in any memory element inside the processor or in the memory area storing the data and the code; we underline the fact that our method addresses, at the same time, faults affecting the data and the code.
- The method is able to guarantee very high fault detection capabilities.
- The area cost of the I-IP is relatively reduced and the memory and performance overheads are, in general, significantly smaller than the ones required by purely software-based approaches.

References

[1] Y. Zorian, "What Is an Infrastructure IP?" *IEEE Design and Test of Computers,* vol. 19, no. 3, pp. 5-7, May/June 2002.

[2] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "Experimentally Evaluating an Automatic Approach for Generating Safety-Critical Software with Respect to Transient Errors," *IEEE Trans. Nuclear Science,* vol. 47, no. 6, pp. 2231-2236, Dec. 2000.

[3] 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 and Distributed Systems,* vol. 10, no. 6, pp. 627-641, June 1999.

[4] N. Oh, P.P. Shirvani, and E.J. McCluskey, "Control-Flow Checking by Software Signatures," *IEEE Trans. Reliability,* vol. 51, no. 2, pp. 111-122, Mar. 2002.

[5] E. Dupont, M. Nikolaidis, and P. Rohr, "Embedded Robustness IPs for Transient-Error-Free ICs," *IEEE Design and Test of Computers,* vol. 19, no. 3, pp. 56-70, May/June 2002.

[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," *J. Electronic Testing: Theory and Applications,* vol. 18, no. 3, pp. 261-271, June 2002.

[7] http://www.eembc.org, 2004.

[8] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "Soft-Error Detection Using Control Flow Assertions," *Proc. IEEE Int'l Symp. Defect and Fault Tolerance in VLSI Systems,* pp. 581-588, 2003.

[9] B. Randell, "System Structure for Software Fault Tolerant," *IEEE Trans. Software Eng.,* vol. 1, no. 2, pp. 220-232, June 1975.

[10] A. Avizienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Trans. Software Eng.,* vol. 11, no. 12, pp. 1491-1501, Dec. 1985.

[11] K.H. Huang and J.A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Trans. Computers,* vol. 33, no. 12, pp. 518-528, Dec. 1984.

[12] 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 and Distributed Systems,* vol. 10, no. 6, pp. 627-641, June 1999.

[13] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques and Tools.* Harlow, U.K.: Addison-Wesley, 1986.

[14] S.S. Yau and F.-C. Chen, "An Approach to Concurrent Control Flow Checking," *IEEE Trans. Software Eng.,* vol. 6, no. 2, pp. 126-137, Mar. 1980.

[15] N. Oh, S. Mitra, and E.J. McCluskey, "ED4I: Error Detection by Diverse Data and Duplicated Instructions," *IEEE Trans. Computers,* vol. 51, no. 2, pp. 180-199, Feb. 2002.

[16] A. Mahmood and E.J. McCluskey, "Concurrent Error Detection Using Watchdog Processors—A Survey," *IEEE Trans. Computers,* vol. 37, no. 2, pp. 160-174, Feb. 1988.

[17] M. Namjaoo and E.J. McCluskey, "Watchdog Processors and Capability Checking," *Proc. Int'l Symp. Fault-Tolerant Computing,* pp. 245-248, 1982.

[18] A. Mahmood, D.J. Lu, and E.J. McCluskey, "Concurrent Fault Detection Using a Watchdog Processor and Assertions," *Proc. IEEE Int'l Test Conf.,* pp. 622-628, 1983.

[19] A.A. Al-Yamani, N. Oh, and E.J. McCluskey, "Performance Evaluation of Checksum-Based ABFT," *IEEE Defect and Fault Tolerance in VLSI Systems,* pp. 461-466, 2001.

[20] M.A. Schuette and J.P. Shen, "Processor Control Flow Monitoring Using Signatured Instruction Streams," *IEEE Trans. Computers,* vol. 36, no. 3, pp. 264-276, Mar. 1987.

[21] M. Namjoo, "CERBERUS-16: An Architecture for a General Purpose Watchdog Processor," *Proc. IEEE Int'l Symp. Fault-Tolerant Computing,* pp. 216-219, 1983.

[22] K. Wilken and J.P. Shen, "Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Control Errors," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 9, no. 6, pp. 629-641, June 1990.

[23] J. Ohlsson and M. Rimen, "Implicit Signature Checking," *Proc. IEEE Int'l Symp. Fault-Tolerant Computing,* pp. 218-227, 1995.

[24] L. Bolzani, M. Rebaudengo, M. Sonza Reorda, F. Vargas, and M. Violante, "Hybrid Soft Error Detection by Means of Infrastructure IP Cores," *Proc. IEEE Int'l On-Line Testing Symp.,* pp. 79-84, 2004.

[25] P. Bernardi, L. Bolzani, M. Rebaudengo, M. Sonza Reorda, F. Vargas, and M. Violante, "Hybrid Soft Error Detection by Means of Infrastructure IP Cores," *IEEE Proc. Int'l Dependable Computing and Comm. Symp.,* pp. 50-58, 2005.

[26] M. Nicolaidis, "Time Redundancy Based Soft-Error Tolerance to Rescue Nanometer Technologies," *Proc. IEEE VLSI Test Symp.,* pp. 86-94, 1999.

[27] R. Velazco, S. Rezgui, and R. Ecoffet, "Predicting Error Rate for Microprocessor-Based Digital Architectures through C. E. U. (Code Emulating Upsets) Injection," *IEEE Trans. Nuclear Science,* vol. 47, no. 6, pp. 2405-2411, Dec. 2000.

[28] F.L. Vargas and M. Nicolaidis, "SEU-Tolerant SRAM Design Based on Current Monitoring," *Proc. IEEE Int'l Symp. Fault-Tolerant Computing,* pp. 106-115, 1994.

[29] J. Browning, R. Koga, and W.A. Kolasisnki, "Single Event Upset Rate Estimates for a 16-K CMOS RAM," *IEEE Trans. Nuclear Science,* vol. 32, no. 6, pp. 4133-4139, Dec. 1985.

Paolo Bernardi received the MS degree in computer science engineering from Politecnico di Torino in 2002. He is currently working toward the PhD degree in computer engineering at the same institution. His main interests include fault-tolerant systems, SoC testing, and diagnosis. He is a student member of the IEEE.

Leticia Maria Veiras Bolzani received the computer science degree from the Universidade Federal de Pelotas—UFPel—Brazil in 2002 and the MSc degree in electrical engineering from Pontificia Universidade Catolica do Rio Grande do Sul—PUCRS—Brazil in 2004. She is currently in Italy, at the Politecnico di Torino, working toward the PhD degree in computer science and automation. Her main research interests include fault-tolerant systems. She is a student member of the IEEE.

Maurizio Rebaudengo received the MS degree in electronics (1991) and the PhD degree in computer science (1995), both from the Politecnico di Torino, Torino, Italy. Currently, he is an associate professor in the Department of Computer Engineering at the same institution. His research interests include testing and dependability analysis of computer-based systems. He is a member of the IEEE.

Matteo Sonza Reorda received the MS degree in electronics (1986) and the PhD degree in computer engineering (1990) from the Politecnico di Torino, Italy. Currently, he is a full professor in the Department of Computer Engineering at the same institution. His main research interests include testing and fault-tolerant design of electronic eystems. He has published more than 200 papers on these topics. He has been the general (1998) and program cochair (2002, 2003) of the IEEE International On-line Testing Symposium. Currently, he is the chair of the committee on Test Program Generation of the IEEE DATE 2004 conference. He is a member of the IEEE.

Fabian Luis Vargas received the BS degree in electrical engineering from the Catholic University (PUCRS), Brazil, in 1988, the MSc degree in computer science from the Federal University of Rio Grande do Sul (UFRGS), also in Brazil, in 1991, and the PhD degree from the Institut National Polytechnique de Grenoble (INPG), France, in 1995. Since 1996, he has been an associate professor at the Catholic University (PUCRS). In 1997, he founded and chaired for six years the Latin American Group of the IEEE Computer Society-Test Technology Technical Council (TTTC). His research interests include fault-tolerant systems design, online testing, and reliability-driven hardware-software partitioning. Professor Vargas is a member of the IEEE and an IEEE Computer Society Golden Core Member.

Massimo Violante received the MS degree in computer engineering (1996) and the PhD degree in computer engineering (2001) from the Politecnico di Torino, Italy. Currently, he is an assistant professor in the Department of Computer Engineering at the same institution. His main research interests include design, validation, and test of fault-tolerant electronic systems. He is a member of the IEEE.