Short Papers

A Hybrid Approach for Detection and Correction of Transient Faults in SoCs

P. Bernardi, L.M. Bolzani Poehls, M. Grosso, and M. Sonza Reorda, *Member*, *IEEE*

Abstract—Critical applications based on Systems-on-Chip (SoCs) require suitable techniques that are able to ensure a sufficient level of reliability. Several techniques have been proposed to improve fault detection and correction capabilities of faults affecting SoCs. This paper proposes a hybrid approach able to detect and correct the effects of transient faults in SoC data memories and caches. The proposed solution combines some software modifications, which are easy to automate, with the introduction of a hardware module, which is independent of the specific application. The method is particularly suitable to fit in a typical SoC design flow and is shown to achieve a better trade-off between the achieved results and the required costs than corresponding purely hardware or software techniques. In fact, the proposed approach offers the same fault-detection and -correction capabilities as a purely software-based approach, while it introduces nearly the same low memory and performance overhead of a purely hardware-based one.

Index Terms—Fault tolerance, SoCs, transient faults, online test.

1 Introduction

NOWADAYS, an increasing number of critical applications are based on Systems-on-Chip (SoCs). Unfortunately, mainly due to the high integration level, this kind of circuits may be particularly sensitive to transient faults, which may possibly cause failures. In this scenario, new techniques that are able to *detect*, *locate*, and *correct* the effects of transient faults in SoCs are fundamental.

Many techniques have been proposed in the past in order to provide the mentioned capabilities to processor-based SoCs, while minimizing insertion cost and system performance reduction. These techniques may be classified as fault detection or fault tolerance techniques depending on the capabilities they provide. The first ones can simply signal the occurrence of a deviation in the normal behavior, and the second ones guarantee that the system continues to work properly even if an error occurs.

Several techniques have been proposed to achieve transient fault detection, which can be further categorized into software-based, hardware-based, and hybrid solutions.

Typically, *hardware-based fault detection* approaches consist in adding special purpose modules, named watchdogs, working in parallel with the system microprocessor [1]. In general terms, these modules can perform three basic types of monitoring operations: memory-accesses check [1], consistency check [2], control flow check [3], [19], [20], or symptoms detection [22]. In general, detection capabilities form an additional area that requires attention.

When dealing with SoC systems, hardware-based fault-detection techniques may effectively be implemented by introducing new modules especially devoted to dependability enhancement. Following this strategy, a new family of Intellectual Property cores (or IP cores), named Infrastructure IP cores (or I-IPs), has been introduced not only to guarantee dependability, but also to support a wide set of possible goals, such as test, silicon debug, diagnosis, etc. [17].

Further, *software-based fault detection* solutions rely on modifications of the embedded software. Among the most important solutions proposed in the literature, there are the Enhanced Control Flow Checking Using Assertions (ECCA) and the Control Flow Checking by Software Signatures (CFCSS) [15], [16]. Other techniques harden SoCs against faults affecting the data they elaborate. An example of such a group of techniques is given by the method proposed in [4], which is based on instruction-level duplication and is able to achieve detection of the data faults. Another approach, named Error Detection by Data Diversity and Duplicated Instruction (ED4I), has been proposed in [5].

Some solutions combining hardware and software techniques have been proposed. These solutions, usually referred to as *hybrid fault detection* solutions [13], [21], can achieve a better trade-off between fault coverage and area as well as performance overheads. A hybrid approach typically introduces some reduced redundant information in the application code and data, while resorting to an I-IP to perform on-the-run consistency checks.

When moving to transient fault correction, the correction is normally obtained by means of error-tolerant structures or through recovery schemes. *Hardware-based fault tolerance* approaches fall in the former category. One of the most important and used solutions is the N-Modular Redundancy (NMR) technique, from which descends the idea of the Triple Modular Redundancy (TMR) [6], [23]. Conversely, *software-based fault tolerance* techniques are based on the execution of additional code to recover the erroneous situation. An important group of solutions is known as Software-Implemented Hardware Fault Tolerance (SIHFT) techniques [6]; the recovery blocks [8], N-version programming [9], algorithm-based fault tolerance (ABFT) [10], and checkpointing [11], [12] are some SIHFT methodologies proposed in the literature. However, their adoption is often limited by the high overhead introduced in terms of increased code size and decreased performance, especially when recovery is needed.

This paper presents a hybrid fault tolerance approach for Systems-on-Chip that extends the concept of hybrid fault detection to fault tolerance. Single Event Upset (SEU) faults [6] in the data memories are detected and corrected by a combination of software- and hardware-based fault tolerance techniques; the approach consists of a set of embedded software modification rules aimed at introducing redundant information in the high-level application code, whose consistency is then checked by an I-IP during the application execution.

This paper describes the I-IP structure and its role in both the detection and correction processes; furthermore, it describes the required software modification rules and the system level software recovery scheme.

The proposed solution leads to significant advantages:

- The ability to supply fault tolerance with a relatively limited decrease of the performance with respect to the solution only providing detection capabilities.
- The ease of integration in current SoC industrial design flows, since the proposed approach is based on high-level software modifications and on the addition of a hardware module without modification to any other hardware modules.
- The capability of managing SoC schemes with cache memories which are integrated into the processor IP and tackling many processor architectures including pipelined ones.

- *P. Bernardi, M. Grosso, and M. Sonza Reorda are with the Dipartimento di Automatica e Informatica, Politecnico di Torino, Corso Duca Degli Abruzzi, 24, 10129 Torino, Italy. E-mail: {paolo.bernardi, michelangelo.grosso, matteo.sonzareorda}@polito.it.*
- *L.M. Bolzani Poehls is with the Engineering Faculty, Catholic University of Rio Grande do Sul (PUCRS), Avenida Ipiranga 6681, Partenon, Porto Alegre (RS), CEP: 90619-900, Brazil. E-mail: leticia.poehls@pucrs.br.*

Manuscript received 14 July 2009; accepted 27 Jan. 2010; published online 6 Aug. 2010.
For information on obtaining reprints of this article, please send e-mail to tdsc@computer.org, and reference IEEECS Log Number TDSC-2009-07-0102. Digital Object Identifier no. 10.1109/2010.33.

```
(1)  a = 2;
(2)  b = 5;
(3)  c = 0;
(4)  while (c < 10) {
(5)        a = a + b;
(6)        c = c + 1;
(7)  }
```

Fig. 1. Unhardened code.

Experimental results show the feasibility and effectiveness on two sample SoCs: the first and simplest one is based on an i8051, while the other is based on LEON3, a RISC processor compliant with the SPARC v8 architecture and featuring a seven-stage pipeline, with enabled data cache. On both experimental setups, evaluations were obtained by running a set of benchmark applications; independently of the system, the high-level (C) code of these benchmarks has been modified according to the given modification rules, and then cross-compiled in the selected architecture instruction set. The experimental results show that in most cases the method is able to provide a very high fault tolerance nearly at the same cost (in terms of size of the I-IP, code size, and execution time) of methods such as [21], which only provides detection capabilities. As a major conclusion, we claim that the new method has the same detection and correction capabilities of pure SIHFT techniques [14], while showing reduced code and performance overheads. Further, the I-IP cost in terms of area overhead is slightly increased with respect to [14], and negligible with respect to the total SoCs area.

The paper is organized as follows: Section 2 presents the background regarding hybrid fault-detection techniques. Section 3 details the proposed hybrid fault tolerance approach. Section 4 describes its implementation of the approach proposed, and in Section 5, some experimental results assessing the costs and benefits of the proposed approach are summarized and analyzed. Finally, in Section 6, some conclusions are drawn.

2 Previous Works

The ideas illustrated in the paper stem from two previous contributions of the authors on hybrid fault detection [21] and software-based fault correction [14] schemes. Fig. 1 shows an example that will be used to summarize the major ideas behind these approaches.

In [21], a hybrid fault detection approach is illustrated; some high-level code modification rules are introduced, mandating the duplication of every variable and of all instructions acting on it: Fig. 2 shows the code of Fig. 1 hardened according to these rules. At the same time, an I-IP is introduced, which is responsible for the consistency check between the two replicas of each variable. In particular, the I-IP monitors the bus looking for bus cycles accessing two replicas of the same variable and checks whether their value is identical. When an inconsistency is observed, the I-IP activates an ERROR signal, which can be sent to the microprocessor or elsewhere. This structure is shown in Fig. 3.

```
(1)  a0 = 2; a1 = 2;
(2)  b0 = 5; b1 = 5;
(3)  c0 = 0; c1 = 0;
(4)  while (c0 < 10) {
(5)        a0 = a0 + b0; a1 = a1 + b1;
(6)        c0 = c0 + 1; c1 = c1 + 1;
(7)  }
```

Fig. 2. Code hardened according to the hybrid fault detection approach proposed in [21].

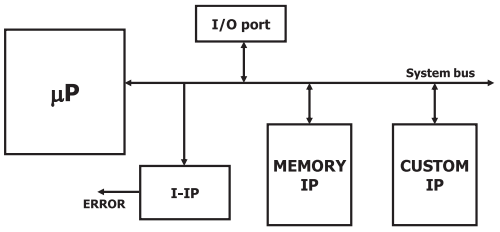

Fig. 3. System architecture with the I-IP proposed in [21].

In this way, the SoC is hardened by implementing the capability to detect faults affecting data memory. In more detail, the consistency check is performed in hardware, which consequently simplifies the hardened code and decreases the memory and performance overheads in respect to software-based techniques such as [5], [9]; the hardened code is simplified and the memory and performance overheads decrease.

In [14], a purely software-based fault-tolerant technique based on a set of transformation rules to be applied to the high-level application code was proposed for tolerating SEUs in the data memory. The technique exploits the same idea as behind the approach presented in [21], since it is based on variable duplication and consistency check operations. However, the approach presented in [14] implements the consistency check operations in software. Further, it implements error correction by using a dedicated variable named *execution checksum*, which is updated during the application's execution, and a software procedure that is able to calculate a *reference checksum* when invoked.

The hardened code is obtained by applying the following high-level transformations rules:

- Every variable x is duplicated, leading to two copies *x0* and *x1*. Two sets of variables are generated, *set0* and *set1.*
- Every write or read operation performed on *x0* must be performed on *x1*, too.
- After every couple of write operations, the consistency of *x0* and *x1* is checked. If an inconsistency is detected, an error recovery procedure is activated.
- The *execution checksum* variable *chk* is associated with one set of variables (let us suppose that it is *set0*). Initially, the value of *chk* is set to 0, while during the application execution it is updated to continuously hold the exclusive-or ($\wedge$) value of all the variables. This can be obtained by updating *chk* before and after every write operation on x, e.g., by executing $chk = chk \wedge x0$ before the write operation on x (*x0* being the old value of the variable) and $chk = chk \wedge x0'$ after it (*x0'* being the new value).
- Software fault detection is obtained by checking the coherence between variable replicas, while software fault tolerance is obtained by resorting to the checksum values. If an error is detected on a variable, the checksum value permits us to identify which of the two replica sets is corrupted: the current value of the checksum is computed for *set0* (*reference checksum*), and compared with the *execution checksum* value stored in chk. If the recomputed checksum matches the stored one, *set0* is fault free, and the fault affected the other set: therefore, values from *set0* can be copied to the corresponding variables in *set1* to correct the error.

The *f_chk()* scans the memory space for the addresses corresponding to the boundaries of the selected replica used to compute the *reference checksum*; in software, this procedure is implemented with a *while* cycle that sequentially accesses the data memory and for each word w_j incrementally performs the

```
(1)  chk = chk^a0;
(2)  a0 = 2; a1 = 2;
(3)  chk = chk^a0;
(4)  chk = chk^b0;
(5)  b0 = 5; b1 = 5;
(6)  chk = chk^b0;
(7)  chk = chk^c0;
(8)  c0 = 0; c1 = 0;
(9)  chk = chk^c0;
(10) while (c0 < 10) {
(11)      chk = chk^a0;
(12)      a0 = a0 + b0; a1 = a1 + b1;
(13)      chk = chk^a0;
(14)      if ((a0 != a1) || (b0 != b1)) {
(15)          if (f_chk() == chk) {
(16)              a1 = a0; b1 = b0;
(17)          }
(18)          else {
(19)              a0 = a1; b0 = b1;
(20)          }
(21)      }
(22)      chk = chk^c0;
(23)      c0 = c0 + 1; c1 = c1 + 1;
(24)      chk = chk^c0;
(25)      if (c0 != c1) {
(26)          if (f_chk() == chk)
(27)              c1 = c0;
(28)          else
(29)              c0 = c1;
(30)      }
(31) }
```

Fig. 4. Code hardened according to the purely software-based fault tolerance approach.

operation $chk = chk \wedge w_j$. Please note that the checksum technique considered here (based on the exclusive-OR operation) is only one of the many existing in the literature [12], [13]. Fig. 4 reports the sample code hardened according to the described rules. The checksums are associated with *set0*.

3 Proposed Approach

This section aims at describing a hybrid approach for Systems-on-Chip that guarantees *tolerance* of transient faults in data memories and caches. Roughly speaking, fault detection is obtained by variable and code duplications as well as consistency checking, while fault location is supported by checksum verification and correction is achieved by a recovery schema.

The method is based on some high-level software modification rules and on the inclusion of a hardware module (or I-IP) connected to the system bus. Similar to approaches described in previous works, the I-IP is responsible for performing in hardware some of the needed computations, thus improving performance and reducing the overhead. The methodology is completed by a fast system-level recovery procedure activated through an interrupt request following detection and again supported by the I-IP.

In summary, the fault-tolerant SoC schema consists in:

1. monitoring the running application by continuously performing consistency checks and updating the execution checksum,
2. stopping the application execution as soon as an error is detected by raising an interrupt request, and
3. recovering the fault effects by activating an interrupt routine that
   a. computes the reference checksum,
   b. compares the reference and the execution checksums for identifying the corrupted replica set,
   c. corrects the corrupted values using the fault-free ones,
   d. restores the correct execution checksum value,
   e. returns the control to the suspended application.

```
(1)  IIPchk(a0);
(2)  a0 = 2; a1 = 2;
(3)  IIPchk(b0);
(4)  b0 = 5; b1 = 5;
(5)  IIPchk(c0);
(6)  c0 = 0; c1 = 0;
(7)  while (c0 < 10) {
(8)      IIPchk(a0); // sends to IIP the a0 value
(9)      a0 = a0 + b0; a1 = a1 + b1;
(10)     IIPchk(c0); // sends to IIP the c0 value
(11)     c0=c0+1; c1=c1+1;
(12) }
```

Fig. 5. Code hardened according to the hybrid fault tolerance approach.

The proposed methodology copes with single processor-based SoCs, independently of the processor type (CISC or RISC), and also with enabled data caches.

3.1 Software Modification Rules

Two high-level code modification rules are required to implement the proposed hybrid scheme.

- Rule 1 is aimed at providing fault-detection capabilities; it requests variable and instruction duplications, while the consistency check is performed by the I-IP, as proposed in [21].
- Rule 2 addresses fault localization by the insertion of extra instructions aimed at updating the execution checksum value; an *IIPchk(x)* macro is included before write instructions on every variable x belonging to the replica set to which the checksum refers. The macro sends the current value of x to the I-IP, which uses it to update the checksum.

Fig. 5 shows the sample code modified according to the proposed approach. According to rule 1 the code is duplicated. Since the consistency check between the two replicas of the duplicated variables is performed by the I-IP for detection, a mechanism is needed to make the I-IP able to quickly identify the values of two replicas of the same variable when traveling on the observed bus. To address this issue, the memory area storing the data is divided into two parts, and each of the two sets of variables is stored in a different part of the data memory [21]. By assigning two starting addresses to the two parts that only differ by 1 bit, the addresses of the two replicas of the same variable always differ by 1 bit, too.

In Fig. 6, an example of data organization is reported: the data segment includes three variables a, b, and c, duplicated in a0, a1, b0, b1, c0, and c1 that are distributed over two segments at addresses differing by 1 bit only. The *Base Address* corresponds to the initial

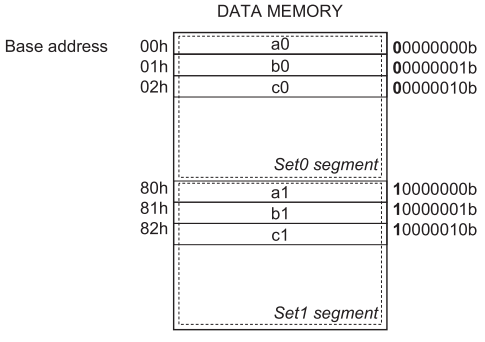

Fig. 6. Data memory organization.

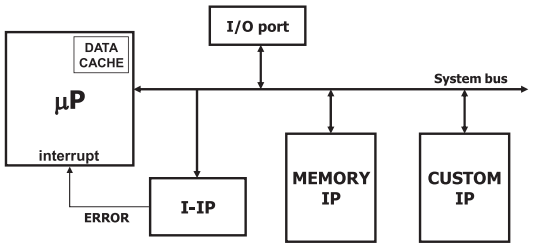

Fig. 7. Fault-tolerant system schematic view.

address of the first part and the *Segment Size* is the size of each part of the area (128 bytes). Therefore, when *Address* is observed on the bus and if $Address < Base\ Address + Segment\ Size$, the address corresponds to the first replica of a variable. Otherwise, the address corresponds to the second replica. The expression $Address1 = Address2 - Segment\ Size$ is thus hardwired in the I-IP for quick replica identification.

According to the second rule, suitable high-level macros named *IIPchk()* are introduced in the application code in order to support the *execution checksum* update operation. Before each write operation on a variable V, the processor executes an *IIPchk(V)* macro to send the variable V's old value to the I-IP. Differently from [14], the new value V is not sent directly to the I-IP by the code, since it is expected to pass on the bus and to be recognized automatically by the I-IP.

3.2 The Infrastructure-IP

The I-IP is an ad hoc module connected to the SoC system bus, which executes the following main tasks:

- consistency check between the two replicas of each variable,
- storage and update of the checksum value during the execution of the application code.

As shown in Fig. 7, the I-IP monitors the system bus connecting the processor to the other system modules. Apart from monitoring the bus, the I-IP is connected to the bus as a standard I/O peripheral interface. This allows the processor to read from and to write to selected I-IP internal registers. This feature is exploited by the *IIPchk()* macro, which writes a value in a suitable register within the I-IP.

The I-IP, whose internal structure is depicted in Fig. 8, monitors the system bus in order to identify the addresses and data values associated with each write cycle. The observed data enable the processor to access some I-IP memory elements, since it is associated with a specific address space. In more detail, the address space is noncacheable, as it is normally the case for addressed correspondence between I/O devices.

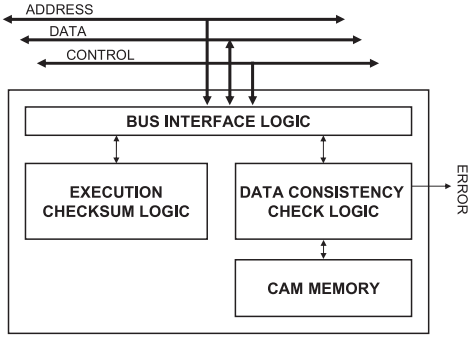

Fig. 8. Block diagram of the I-IP.

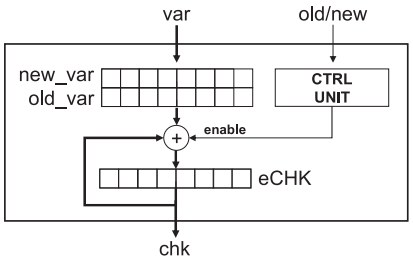

Fig. 9. Internal structure of execution checksum logic.

For the sake of detection, when a write cycle is observed, the *Bus Interface Logic* captures the address as well as the data value on the bus and sends this couple (*adx, value*) to the *Data Consistency Check Logic*, which is supported by a *Content Addressable Memory (CAM)*. Functionally, the I-IP implements the following fault detection algorithm:

- It observes all the write operations taking place on the bus, capturing both address and data values.
- If the captured values are associated with the first replica of a variable, a new entry is inserted in an internal CAM of the I-IP.
- If the captured values are associated with the second replica of a variable, a check is performed by accessing to the internal CAM of the I-IP: if the first replica is not found, or the two replicas do not hold the same value, an ERROR signal is activated.

In order to support fault tolerance, the I-IP contains the *Execution Checksum Logic* that stores the *execution checksum* value in a suitable register and takes care of its continuous update along the application execution: it receives the value of a variable sent by the *IIPchk()* macro immediately before a write operation and captures the new value of the variable available on the bus while the write operation is performed.

Fig. 9 illustrates the structure of the *Execution Checksum Logic*: the old and new values of the write variable are first stored in a couple of internal registers, which are added to the execution checksum value only when both have been received under the control of a control unit; this architecture guarantees the correctness of the recovery procedure.

3.3 The Recovery Procedure

A software function (hereinafter named *Data_recovery()*) provides correction capabilities to the SoC. When an error is detected by the I-IP, the ERROR signal is raised that may trigger an interrupt request. Due to this external event the application code is suspended and the *Data_recovery()* interrupt service routine is launched. Fig. 10 reports a timing diagram showing the several events related to the activation, execution, and the end of the *Data_recovery()* function.

Fig. 11 shows the *Data_recovery()* C code supposing that the execution reference is computed by the I-IP on *set0*. Once activated, the *Data_recovery()* function first stops the monitoring capability of

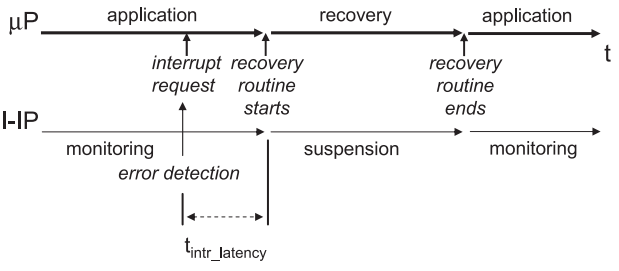

Fig. 10. Recovery procedure timing diagram.

```
(1)  procedure data_recovery () {
(2)     Suspend_monitoring();
(3)     if (Calculate_ref_chk(set0) == IIPread_exec_chk()){
(4)           Restore_variables(set0);
(5)           IIPwrite_exec_chk(Calculate_ref_chk(set1));}
(6)     else
(7)           Restore_variables(set1);
(8)     Invalidate_cache();
(9)     Restart_monitoring();
(10)    return;}
```

Fig. 11. Code of the recovery procedure.

the I-IP (line 2) and computes the reference checksum value for *set0* by reading the main memory to access the complete set of variables associated with *set0* (line3); by comparing the reference checksum and the execution one, read from the I-IP (line 3), it identifies the faulty variable set and performs a selective correction (lines 4 and 7). In case *set0* was faulty, the execution checksum value in the I-IP is recomputed (line 5). Immediately before concluding the recovery procedure, if a data cache is present and enabled, its content is invalidated (line 8) because it might possibly be corrupted, and the I-IP monitoring functionalities are restarted (line 9). When the recovery procedure execution ends, the application code restarts from the point where it was suspended.

It is worthwhile to note that thanks to the Execution Checksum Logic structure waiting for reception of both old and new values to update the checksum as discussed previously, the proposed methodology is independent of the recovery routine launch latency that varies from processor to processor ($t_{intr\_latency}$ in Fig. 10). The latency time, e.g., due to the execution of instructions already in the pipeline, is defined as the interval between the real error occurrence and the raising of the error signal, which leads to the start of the recovery procedure. During this interval, in some cases, the value of the old replica is sent to the I-IP contributing to the checksum's computation before recovery starts. Therefore, the corresponding new one will be read from the bus only after the recovery procedure ends. This particular situation results in the execution checksum being partially updated; therefore, its value is inconsistent with the memory content. The circuitries described in the previous paragraph and shown in Fig. 9 aim at avoiding this critical condition.

3.4 Further Software Optimizations

By analyzing the high-level modifications associated with the proposed approach, it is possible to apply a set of optimizations to the proposed method in order to reduce its impact on the performance and code overheads. According to the transformation rules described in Section 3.1, the activation of the *IIPchk()* macro is introduced before each write operation existing in the application code. However, it is possible to identify some situations where this operation is not necessary. In particular, during the execution of the program code the first replica of each variable associated with a read or write operation is temporarily stored in the CAM and remains in the I-IP waiting for the second replica that will pass through the bus. After the consistency check is performed, these values remain in the CAM until they are possibly replaced by other variables. This means that when a write operation is performed on a variable, there are cases where a previous value of the variable is already stored in the CAM, and the *IIPchk()* is not necessary. In the extreme case, the CAM may be sufficient to store all variables manipulated by the program during its execution time.

It is important to note that the decision whether to introduce the *IIPchk()* instruction or not can be taken at compile time based on the analysis of the program code and on the knowledge about the size of the CAM. Thus, the compiler decides whether the insertion of the *IIPchk()* is needed or can be avoided.

3.5 Detection Capabilities

The detection capabilities for single transient faults affecting variables stored in the main memory are guaranteed by the approach, as already discussed in [21]. Such an ability is not affected by the usage of data caches, provided that they adopt the write-through technique, which assures that the content of the cache memory and the main memory is always consistent. Under this assumption, the activation of the data cache does not change the sequence of processor write operations.

In addition, single transient faults affecting the data cache memory can be detected and corrected. The following faulty cache configurations may occur when a fault affects an entry of the data cache (examples refer to Fig. 5):

- The corrupted data cache line is not used; in this case, the fault is effectless or silent; otherwise
- The value of a stored variable is corrupted, then
  - used to compute another variable (such as variable *b1* when used to calculate *a1* at line 9); this fault is indirectly detected when the computed variable value is written in the memory.
  - used to update its own value (such as variable *a1* when used to calculate *a1* again at line 9); in this case, the fault is directly detected when the updated value is written in the memory.
- A cache tag is corrupted, thus originating two situations:
  - The tag is changed in such a way that the cache page storing it is no longer accessed by the processor. This fault will result in a cache miss, which will affect the program performance, only.
  - The tag is changed in such a way that it corresponds to an address that the processor issues during program execution. As a result, the data provided to the processor by the cache differs from the correct one. The I-IP is then likely to observe replica mismatches during the successive bus write cycles.

4 Experimental Results

To evaluate the proposed approach, and in particular to quantify the memory and performance overheads, the high-level code (C language) of the following programs inspired by the EEMBC automotive/industrial suite [18] has been hardened:

- Matrix (MTX): a 10×10 matrices product program.
- Fifth-Order Elliptical Wave Filter (ELPF): an elliptical filter over a set of 16 samples.
- Lempel-Ziv-Welch Data Compression Algorithm (LZW): compresses data by replacing strings of characters with single codes.

These programs have been implemented in the following versions:

- PLAIN: application code without hardening,
- H_DETECT: hybrid fault-detection version according to [21],
- S_CORRECT: pure software-based fault tolerance version according to [14], and
- H_CORRECT: code hardened according to the proposed hybrid correction approach.

These benchmarks were cross-compiled for running on two SoCs equipped with an I-IP designed as proposed in the previous sections: the processor in the first SoC is an Intel 8051 8-bit microcontroller, while in the second a LEON3 is included with enabled data cache memories.

For both cases, the approach's feasibility and effectiveness have been proven and the following parameters were computed:

TABLE 1
Equivalent Gates of the I-IP for Hybrid Detection and Correction

Logic component	Equivalent gates [#]
Bus Interface Logic	251
Data Consistency Check	1,348
Checksum Logic	972
CAM	1,736
Total I-IP	**4,307**

- *area overhead* introduced by the I-IP in the SoC,
- *memory overhead* deriving from code duplication as well as from the introduction of the calls to *IIPchk()*, and
- *performance overhead* in terms of additional execution time with respect to the plain version.

4.1 Results Achieved on the i8051 SoC

To quantify the area overhead introduced by the adoption of the I-IP, a VHDL description of the processor has been considered and then synthesized using a commercial tool (Synopsys Design Compiler) in a generic library. The size of the CAM embedded in the I-IP has been set to 16 couples (*adx, value*). Table 1 summarizes the number of equivalent gates of the I-IP. In the hardened system, the area overhead introduced by the adoption of the I-IP is less than five percent of the total SoC area.

It should be noted that the size of the I-IP is only slightly higher (i.e., a few hundreds of equivalent gates) than that of a corresponding I-IP which provides detection capabilities only, whose characteristics are reported in [21].

Table 2 summarizes the data and code memory occupation of the four programs implemented in the versions described above. The memory occupation regarding the data and the code segments has been measured in bytes. The code overhead introduced by the approach proposed in this paper is almost the same as the one introduced by the approach proposed in [21], which is only able to detect possible faults affecting the data. Moreover, the proposed approach introduces a code overhead significantly smaller than the one of the purely software solutions [14]. Table 3 summarizes the performance overhead of the four versions of each program.

The reported data show that the proposed technique allows us to reduce the performance overhead in all the considered test cases with respect to other fault tolerant approaches, such as the one proposed in [14]. In all the considered versions, the final performance overhead is nearly the same as that of the method proposed in [21], demonstrating once more that the proposed method introduces fault tolerance capabilities without any significant penalty with respect to the pure fault-detection approach.

TABLE 2
Code and Memory Overheads

Program	Version	Code size [byte]	Code size Overhead [%]	Data size [byte]	Data size Overhead [%]
MTX	PLAIN	290	-	102	-
	H_DETECT	541	86.5	204	100.0
	S_CORRECT	942	224.8	205	101.2
	H_CORRECT	**696**	**140.0**	**204**	**100.0**
ELPF	PLAIN	244	-	34	-
	H_DETECT	466	91.0	68	100.0
	S_CORRECT	2,429	895.5	69	102.9
	H_CORRECT	**682**	**179.5**	**68**	**100.0**
LZW	PLAIN	232	-	35	-
	H_DETECT	789	240.1	70	100.0
	S_CORRECT	2,116	812.1	71	102.8
	H_CORRECT	**831**	**258.2**	**70**	**100.0**

TABLE 3
i8051 Performance Overhead

Program	Version	Execution time [clk cycles]	Execution time Overhead [%]
MTX	PLAIN	110,508	-
	H_DETECT	210,335	85.4
	S_CORRECT	330,678	192.6
	H_CORRECT	**220,387**	**99.4**
ELPF	PLAIN	14,346	-
	H_DETECT	21,987	53.3
	S_CORRECT	42,102	193.5
	H_CORRECT	**22,716**	**58.3**
LZW	PLAIN	19,209	-
	H_DETECT	34,289	78.5
	S_CORRECT	47,338	146.4
	H_CORRECT	**34,975**	**82.1**

The number of clock cycles required to recover the data memory content has not been considered, since we assume that the typical SEU occurrence frequency is rather low.

4.2 Results Achieved on the LEON3 SoC

To demonstrate the applicability and effectiveness of the solution in terms of reusability, we simply recompiled without any modification the C application code used for the Intel 8051 in the SPARC v8-compliant LEON3 instruction set. The performance overhead is shown in Table 4. Beyond the PLAIN and S_CORRECT, results are also provided for the ED^{4}I [5] and the ABFT [10] (for MTX only) techniques.

We also modified the I-IP to plug it into the new system. The size of the I-IP has increased (7,233 equivalent gates) with respect to the i8051 evaluation due to the larger width of the AMBA-AHB bus the I-IP needs to be connected to. However, the percentage area overhead stemming from the I-IP is decreased due to the much larger size of the LEON3 core.

A set of fault injection campaigns was then performed in order to evaluate the proposed methodology resorting to the environment described in [24], where hardware emulation by means of FPGAs is used to speed up the evaluation process. Fault effects observed were classified as follows:

- *Silent*: the fault does not modify the program execution.
- *Exception*: the fault causes a processor exception; for instance, with the corruption of an index used for accessing an array, or a pointer, it is likely that the fault will result in an invalid address exception.
- *Detected*: the fault is detected (and recovered when applies).
- *Wrong Answer*: the output results differ from the expected without any detection.

TABLE 4
LEON3 Performance Overhead

Program	Version	Execution time [clk cycles]	Execution time Overhead [%]
MTX	PLAIN	15,787	-
	S_CORRECT	46,129	192.2
	ED^{4}I	35,631	125.7
	ABFT	43,477	175.4
	H_CORRECT	**33,626**	**113.0**
ELPF	PLAIN	2,049	-
	S_CORRECT	15,957	678.6
	ED^{4}I	6,853	234.4
	H_CORRECT	**3,834**	**87.1**
LZW	PLAIN	2,744	-
	S_CORRECT	16,394	497.4
	ED^{4}I	8,713	217.5
	H_CORRECT	**6,111**	**122.7**

TABLE 5
Results for Faults Affecting the Data Memory

Program	Version	Silent [#]	Exception [#]	Detected [#]	Wrong Answer [#]
MTX	PLAIN	23,430	0	0	6,570
	S_CORRECT	19,950	0	10,050	0
	ABFT	23,610	0	6,390	0
	ED^{4}I	18,300	0	11,700	0
	H_CORRECT	**20,360**	**0**	**9,640**	**0**
ELPF	PLAIN	24,750	0	0	5,250
	S_CORRECT	21,770	0	8,230	0
	ED^{4}I	17,460	0	12,540	0
	H_CORRECT	**21,460**	**0**	**8,540**	**0**
LZW	PLAIN	28,470	60	0	1,470
	S_CORRECT	27,392	58	2,550	0
	ED^{4}I	28,130	90	1,780	0
	H_CORRECT	**27,620**	**57**	**2,323**	**0**

TABLE 6
Results for Faults Affecting the Data Cache

Program	Version	Silent [#]	Exception [#]	Detected [#]	Wrong Answer [#]
MTX	PLAIN	21,714	0	0	8,286
	S_CORRECT	22,650	0	7,350	0
	ABFT	11,319	0	18,678	0
	ED^{4}I	17,148	0	12,852	0
	H_CORRECT	**23,973**	**0**	**8,798**	**0**
ELPF	PLAIN	14,994	0	0	15,006
	S_CORRECT	16,686	0	13,314	0
	ED^{4}I	9,411	0	20,589	0
	H_CORRECT	**19,994**	**0**	**10,006**	**0**
LZW	PLAIN	26,828	183	0	2,989
	S_CORRECT	25,747	154	4,199	0
	ED^{4}I	20,196	138	9,666	0
	H_CORRECT	**25,566**	**142**	**4,592**	**0**

Table 5 reports the results we obtained for faults affecting the memory storing the program data. In the presented benchmarks, data memory almost fits within the processor data cache. As a result, either the fault hits the memory before the variable is loaded into the cache, or it has no effect. For this reason, most of the faults are classified as Silent. The errors that are able to propagate to the data provoking Wrong Answers in the Plain implementation are detected by all the hardening approaches. Few exceptions have been observed as results of erroneous memory accesses by indexes or pointers during LZW execution haltered by the injected faults.

Table 6 reports the results of the fault-injection campaigns targeting the processor data cache memories. As far as concerning the data cache, we observed that the number of Silent faults is reduced with respect to faults affecting the memory, as already discussed before. Moreover, several faults produced Wrong Answers in the PLAIN implementation, while most of them are detected in the hardened versions either by the employed hardening approach, or by the processor exception mechanism.

5 Conclusions

The approach proposed in this paper combines software and hardware to guarantee detection and correction capabilities with respect to transient faults affecting the data manipulated by a SoC. The proposed approach is based on a set of transformation rules to be applied to the high-level code of the application, an ad hoc I-IP, and a recovery procedure. Moreover, some optimizations aiming at the reduction of the impact in terms of the code size and application performance introduced by the proposed approach are described.

The method is able to effectively detect and correct a very large percentage of transient faults that can affect the data, independently of the exact fault location (memory, cache, and registers). Moreover, it is particularly suitable to be included in the typical design flow of current SoCs because it is only based on easily automatable modifications on the high-level code and on the introduction of an application independent I-IP: no changes are required in the processor core. This characteristic permits low design times and a reduced impact on existing modules.

References

[1] 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.

[2] 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.

[3] 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.

[4] O. Goloubeva et al., "Soft-Error Detection Using Control Flow Assertions," *Proc. IEEE Symp. Defect and Fault Tolerance in VLSI Systems,* pp. 581-588, 2003.

[5] 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.

[6] D. Pradhan, *Fault-Tolerant Computer System Design.* Prentice Hall, 1996.

[7] O. Gouloubeva et al., *Software-Implemented Hardware Fault Tolerance.* Springer Science + Business Media, p. 228. 2006.

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

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

[10] K.H. Huang and J.A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Trans. Computers,* vol. 33, no. 6, pp. 518-528, June 1984.

[11] K.M. Chandy and C.V. Ramamoorthy, "Rollback and Recovery Strategies for Computer Programs," *IEEE Trans. Computers,* vol. 21, no. 6, pp. 546-556, June 1972.

[12] J. Long, W.K. Fuchs, and J.A. Abraham, "Compiler-Assisted Static Checkpoint Insertion," *Proc. 22nd Int'l Symp. Fault-Tolerant Computing,* pp. 58-65, 1992.

[13] L. Bolzani, P. Bernardi, and M. Sonza Reorda, "A Hybrid Approach to Fault Detection and Correction in SoCs," *Proc. 13th IEEE Int'l On-Line Testing Symp.,* July 2007.

[14] M. Rebaudengo, M. Sonza Reorda, and M. Violante, "A New Approach to Software-Implemented Fault Tolerance," *The J. Electronic Testing: Theory and Applications,* vol. 20, pp. 433-437, Aug. 2004.

[15] Z. Alkhalifa et al., "Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detecion," *IEEE Trans. Parallel and Distributed Systems,* vol. 10, no. 6, pp. 627-641, June 1999.

[16] N. Oh, P.P. Shirvani, and E.J. McCluskey, "Control-Flow Checking by Software Signatures," *IEEE Trans. Reliability,* vol. 51, no. 2, pp. 111-112, Mar. 2002.

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

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

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

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

[21] P. Bernardi et al., "A New Hybrid Fault Detection Technique for System-on-a-Chip," *IEEE Trans. Computers,* vol. 55, no. 2, pp. 185-198, Feb. 2006.

[22] N.J. Wang and S.J. Patel, "Restore: Symptom Based Soft Error Detection in Microprocessors," *Proc. IEEE Int'l Conf. Dependable Systems and Networks,* pp. 30-39, 2005.

[23] J. Gaisler, "A Portable and Fault-Tolerant Microprocessor Based on the SPARC V8 Architecture," *Proc. IEEE Int'l Conf. Dependable Systems and Networks,* pp. 409-415, 2002.

[24] P. Civera et al., "An FPGA-Based Approach for Speeding-Up Fault Injection Campaigns on Safety-Critical Circuits," *The J. Electronic Testing: Theory and Applications,* vol. 18, no. 3, pp. 261-271, June 2002.