On the use of Embedded Debug Features for Permanent and Transient Fault Resilience in Microprocessors

M. Portela-Garcia[a], M. Grosso[b], M. Gallardo-Campos[a], M. Sonza Reorda[b], L. Entrena[a], M. Garcia-Valderas[a], C. Lopez-Ongil[a]

*Corresponding author: (marta.portela@uc3m.es)
[a] University Carlos III of Madrid, Leganés, 28911, Spain
[b] Politecnico di Torino, Torino, Italy

Abstract

Microprocessor-based systems are employed in an increasing number of applications where dependability is a major constraint. For this reason detecting faults arising during normal operation while introducing the least possible penalties is a main concern. Different forms of redundancy have been employed to ensure error-free behavior, while error detection mechanisms can be employed where some detection latency is tolerated. However, the high complexity and the low observability of microprocessors' internal resources make the identification of adequate on-line error detection strategies a very challenging task, which can be tackled at circuit or system level. Concerning system-level strategies, a common limitation is in the mechanism used to monitor program execution and then detect errors as soon as possible, so as to reduce their impact on the application. In this work, an on-line error detection approach based on the reuse of available debugging infrastructures is proposed. The approach can be applied to different system architectures profiting from the debug trace port available in most of current microprocessors to observe possible misbehaviors. Two microprocessors have been used to study the applicability of the solution, LEON3 and ARM7TDMI. Results show that the presented fault detection technique enhances observability and thus error detection abilities in microprocessor-based systems without requiring modifications on the core architecture.

Keywords

Error detection, debug infrastructure, on-line test.

On the use of Embedded Debug Features for Permanent and Transient Fault Resilience in Microprocessors

1. Introduction

One of the challenges [1] in current and future digital circuits is to ensure correct behavior during their normal operation. This is especially true in the case of mission- or safety-critical applications such as medical, avionic and automotive ones, where emerging standards such as [2] demand higher levels of fault detection and tolerance from circuit manufacturers. Reduced transistor sizes, lower operation voltages and higher working frequencies make circuits more sensitive to transient faults, such as soft errors caused by cosmic rays and radioactive contaminants in packages. Also, other kinds of phenomena, like aging and wear-out effects, are becoming an increasing concern, causing both transient and permanent faults during mission time. Therefore, on-line protection techniques against transient and permanent faults are mandatory to ensure suitable levels of dependability.

The increasing device integration level brings about systems with high complexity, like Systems on Chip (SoCs). SoCs may embed one or more microprocessors, memories and other components. In other cases, the same device can accommodate an increasing number of processor cores, which can effectively be used to improve the system performance. In general, microprocessors are main components in current digital circuits and fault protection techniques devised specifically for these kinds of components are needed. In general, protection techniques consist of two main steps: error detection to discover the effects of a fault as soon as it happens, and error recovery, in order to continue with the system execution with the minimum disturbance. However, the complexity of current microprocessors hinders the task of error detection. In particular, behavior monitoring and result inspection are critical issues due to the low observability of internal resources [3].

Traditional methodologies are based on hardware and/or time redundancy and usually require changes in the application code, hardware modifications, or access to inner buses. In order to increase observability for on-line error detection in microprocessors we propose a solution based on profiting from the availability of integrated debugging infrastructures. Modern microprocessors include on-chip debuggers to facilitate the task of designing the software to be run on the system. They provide access to processor registers and memory areas and can include trace interfaces or buffers for tracking program execution. Once an application has been developed and validated on a system, the debugging infrastructures become inactive and useless during mission time. By means of debug interfaces, the behavior of the processor during the execution of whichever piece of code can be observed in details (thus improving the chances of detecting possible faults) without affecting the normal operation, nor involving any significant change in the application code, nor asking for any additional hardware structure.

Some previous works based on this idea were presented in [4] and [5]. The approach presented in [4] is based on the availability of two processors in the system to duplicate the execution and on the reuse of available debugging infrastructures to observe and compare both runs, thus detecting possible misbehaviors. In that paper it was demonstrated that direct monitoring of the trace interface enables same or better fault detection than when observing the data bus. This method is generalized in [5] in order to be used in combination with different hardening techniques, allowing the implementation of different fault protection levels. In this paper, we collect the former results and make a further step in order to widen the generality of the method. A study of the effectiveness of the approach when the trace data is sampled at different instants is presented. This analysis is necessary in order to prove the applicability of this method to microprocessors with an access to a trace data buffer as well as for those with a direct access to the trace bus. Experimental results with two different microprocessor-based systems based on LEON3 and ARM7TDMI, which include different debugging features, are presented. Also, the main issues related to the implementation of our approach on the two test cases are studied in order to assess its applicability.

The paper is organized as follows: Section 2 presents an overview of related works. Section 3 describes the proposed solution to detect faults during normal operation in microprocessor based systems, by means of reusing debugging infrastructures in various system architectures.

Section 4 describes two case studies, based on the LEON3 and the ARM7TDMI cores, respectively, and presents the results related to the different possible implementations of our approach in both cases. Section 5 collects the results of the fault injection experiments aimed at measuring the method capability in detecting errors for different types of implementations. Finally, Section 6 draws some conclusions.

2. Related Works

The main objective of this paper is to propose a widely applicable on-line fault detection technique focused on current microprocessors. A major limitation of the existing techniques is the reduced accessibility to internal resources, which limits the observability of the processor behavior. In this paper this limitation is attacked by profiting from the debugging infrastructures available in most of current microprocessors. In the following, firstly a review of the existing on-line fault detection techniques is presented; secondly, a summary of the capabilities of today's microprocessor debugging infrastructures as well as of the existing works that exploit them to enhance accessibility to embedded processors is given.

Existing on-line fault detection techniques

Among the existing on-line fault detection techniques that can be applied to increase dependability in microprocessor-based systems, different types can be distinguished depending on the system features (single-processor or multi-processor) and on the fault detection requirements (concurrent or non-concurrent). Moreover, the existing techniques can be classified depending on the implementation: software-implemented techniques, which are based purely on software detection mechanisms, and hardware-based ones, which involve circuit modifications.

If concurrent error detection is not required, periodic test application can be used to detect permanent faults that may arise due to external stresses or aging along the product mission time. Hardware-based self-test techniques can be employed, such as approaches based on Built-In Self-Test (BIST) or specific Infrastructure-IPs (I-IPs), also reusing manufacturing test structures, which provide excellent test quality but often require substantial design modifications, high hardware overhead or increase power consumption. Conversely, Software-Based Self-Test (SBST) methodologies exploit the microprocessor itself to perform at-speed test application and response evaluation by running suitably developed test programs [6]. Usually, the test program includes instructions able to excite the faults and usually a compaction routine (e.g., corresponding to a software Multiple Input Signature Register, or MISR, routine), and instructions to transfer the results to memories or externally accessible ports; alternatively, the test program may write partial results to the processor ports with a hardwired MISR compressing the results [3][7].

Concurrent error detection is usually required in mission- or safety-critical applications, where the effects of both permanent and transient faults need to be detected and possibly corrected as soon as possible after their occurrence. For this purpose, designers insert additional code lines or specific hardware structures to store flow or/and elaborated data and check the consistency between the expected and executed values. Employing these techniques, the possibility of having an error condition with possible catastrophic effects is minimized as much as the error coverage is increased and fault latency is reduced. The usual higher implementation costs (e.g., silicon area and code length) are the price to pay to guarantee higher system security.

Common techniques based on software modifications exploit the concepts of information, operation and time redundancy to detect the occurrence of errors during program execution. Traditional software-based techniques demand the replication of the execution of a program and the consequent vote among the results produced by each replica (e.g., Recovery Blocks [8] and N-Version Programming [9]). Although effective, these approaches rely on software designers for their implementation, i.e., the programmers are in charge of devising how to replicate the program and how to realize the voting mechanism that best fits the application the program implements. These processes are, in general, not automated and thus highly error prone. More recent techniques (such as [10] and [11]) harden programs against errors by replicating variables and/or introducing some control instructions. These techniques simplify the task for software designers since some of them can be applied automatically to the software that is intended to be hardened. When replicating variables, beyond the overhead due to the repetition of computing instructions on two sets of data, an additional cost stems from the need to compare the obtained results.

Hardware-based techniques exploit hardware redundancy. Usually, duplication of cores is used for fault detection and triple module redundancy (TMR) for fault correction in presence of single faults [12] (the probability of more than one simultaneous error in different replicas of a core is usually regarded as very low). Special-purpose hardware modules, called watchdog processors [13] are often used to monitor the control-flow of programs, as well as memory accesses. The introduced modules generally observe the processor bus, its ports and/or other internal critical signals, as described in [14] and [15]. These techniques introduce modifications in the processor or system structure that increase the circuit area and may also involve speed and power overheads.

Hybrid methodologies (i.e., including both hardware and software modifications) have also been proposed (e.g., [16], [17]) to find better trade-offs between error detection abilities and application costs.

In multi-core or multi-processor systems, the different processors can be exploited in order to perform fault detection and recovery tasks [18]-[20]. In this case, there are additional problems to consider since the different processors may share resources, and any possible collision must be prevented. In [18] a summary of the most representative error detection and recovery techniques aimed at multi-core systems is presented. For this kind of systems, redundant execution is an attractive solution to detect errors during normal operation since all cores are not always used and unused processors can execute redundant threads.

Enhanced system observation through Debug features

The complexity of today's systems makes validation of hardware and software modules a very demanding operation. Approaches based on detailed circuital simulation are limited by the computational effort needed, while simplified behavioral models may hide hardware effects that will be manifest in the final product. For this reason, significant system validation and software development works are performed directly on silicon prototypes. Post-silicon hardware/software validation in embedded systems is often supported by specific distributed infrastructures enabling additional circuit visibility (probes) or by means of trace buffers or hardware recorders [21][22][23][24].

Most current microprocessors include dedicated logic to support software debugging operations. This circuitry, usually named *On-Chip Debugger* (OCD), allows the designer to control execution and access internal resources from outside the system. Typical capabilities supported by an OCD include breakpoint and watchpoint setting, step execution, access to internal registers and memory, program and data trace. In order to connect a host computer, the debug interface is made available off-chip either directly or through some standard port such as JTAG or BDM. Serial ports are used for simple debugging operations, while parallel ports are used to support data-intensive debug operations such as real-time tracing. The Nexus 5001 Forum™ standard [25] defines a rich set of OCD features divided in 4 classes, where each class is a superset of the features supported by the lower classes. This standard is becoming increasingly adopted; in addition, most of these OCD features can also be found in other microprocessors that do not comply with the standard.

Embedded microprocessor cores usually come along with modules for improved support of debugging functions. Examples of these modules are the ARM Embedded Trace Macrocell (ETM) [26], the Xilinx MicroBlaze™ Trace Core (XMTC) [27], and the LEON3 Debug Support Unit (DSU) [30].

Figure 1 shows the generic architecture of a (possibly multi-core) embedded CPU equipped with an OCD module. Usually, the latter is tightly connected to the processing core(s) through a high bandwidth interface (Debug I/F), which enables control and monitoring operations. Depending on the specific architectures, different information may move on this connection, including the program counter value, the instruction register content (operation code), the results of arithmetic operations, memory operation target addresses and other processor status flags and signals describing the conditions of pipeline and operation units.

The OCD is typically a programmable module whose functions may include:

- storage of trace data in an internal buffer
- compression of program flow information through the detection of alterations in the program run (i.e., jumps in the executed code)
- monitoring and comparison of the observed address or data values with predefined ones so as to trigger breakpoints and watchpoints
- control of microprocessor cores for activating breakpoints and step-by-step execution.

In embedded systems, OCDs (and, when available, trace buffer information) can be accessed through dedicated interfaces or relying on communication peripherals connected to the system bus.

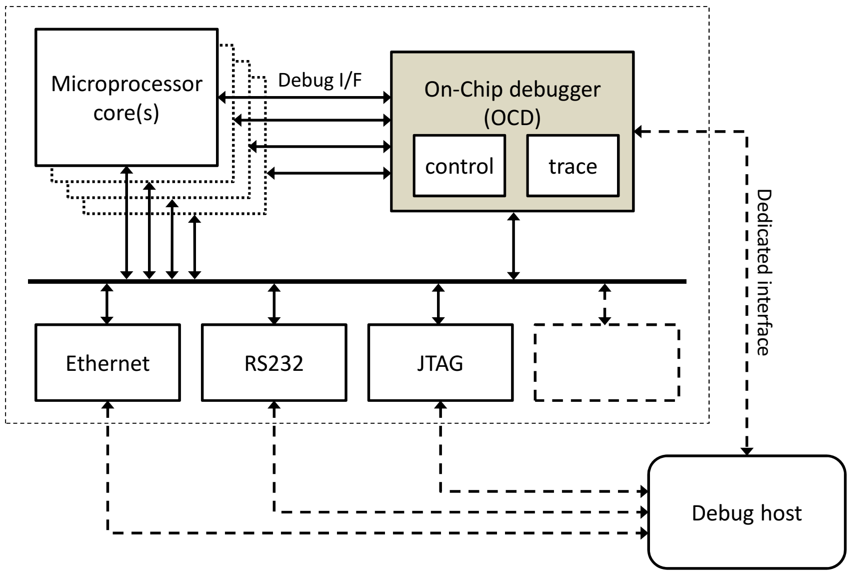

Figure 1. Generic architecture of a CPU equipped with an On-Chip Debugger

Previous works have exploited debug interfaces to access the microprocessor's internal resources for different purposes: the authors in [28] and [29] present fault injection techniques based on using OCDs and show that their features provide minimally intrusive access to embedded cores.

The approach presented in [4] for dual-core systems, then extended in [5] to other system configurations, uses the already available debug infrastructure during normal operation to increase the accessibility to the microprocessor in order to detect faults in a non intrusive way. It takes advantage from the available parallel debug interface, in order to minimize area and performance penalties. Unlike other hardware-based on-line fault detection techniques, in such a solution the debugging interface is observed instead of the main bus, and thus the normal operation is not disturbed. Experimental results show that this technique presents the following advantages with respect to alternative methods:

- it increases the observability during the execution of an on-line test, increasing the fault coverage, also with respect to approaches directly observing the processor bus [4];
- it applies a low cost solution, since it is based on reusing available resources in the system, so as to avoid circuit modification and area overhead;
- it does not affect the performance, as the observation of internal data is done in a non-intrusive way;
- it is non code intrusive. The fault detection is performed by a dedicated hardware module at the expense of a small area overhead, without the need to modify the processor core.

3. On-line Fault Detection Technique

In this work we propose a method that provides a general mechanism to detect the occurrence of permanent and transient faults, causing the least possible disturbance in the microprocessor system. Such technique extends the work presented in [4] and [5] to the most general case. In the proposed method each task considered critical for the application is replicated different times and/or in different processors. The problem of selecting the task(s) to be replicated depends on the required reliability level and on the available resources. The implications of the trade-off between level of protection and costs have been widely debated in literature and are outside the scope of this work. As a worst case, all tasks may be replicated, as in traditional redundancy-based techniques [12]. The task scheduler needs therefore to be programmed accordingly. The several executions of a task are compared relying on data obtained through the debugging infrastructure and possible misbehaviors caused by faults can be detected. Nowadays, most processors include debugging infrastructures; specifically, in this work the trace interface is used, since it is commonly directly accessible in modern processors (Standard Nexus, class 2, 3 and 4, [25]). The information that can be obtained from this trace interface usually includes the value of the program counter, the operation code (*opcode*), instruction results, load data and store data.

Two alternative methods exist to capture trace information: data coming from the processor can be directly intercepted, or they can be temporarily stored in the trace buffer (usually implemented as a circular buffer architecture), which will subsequently be read. The first solution provides a higher degree of fault coverage and minimizes latency, but requires direct access to a large bandwidth interface. Conversely, the buffer solution, to be used when no direct trace observation is available, may introduce additional fault detection latency and may reduce fault coverage, depending on the buffer size and the access policies. As a matter of fact, the task to be monitored may be as long as to make the buffer overflow: in this case, either the task is divided in a series of shorter sub-tasks to be monitored, each one not causing overflow, or only the latest data written in the circular buffer for a task may be employed for checking, therefore reducing the method effectiveness.

In order to implement the method, a specific customizable hardware module can be adopted, which observes the execution of replicated critical tasks. This new hardware module can either be integrated in a digital design without requiring core modifications (in FPGA or ASIC), or be connected to an external trace port: as it will be shown in Section 5, the effectiveness in detecting faults is proportional to the debug interface bandwidth. The module calculates a signature for each replica of the addressed tasks employing the information obtained from the debugging infrastructures. Whenever the tasks finish their execution these signatures are compared and if they differ, then a fault is detected and the hardware triggers some error signal. The fault latency achieved with this approach corresponds to the time that the replicated task takes to finish its execution. By profiting from debugging infrastructures the observability is increased and the costs are low since hardware from the original component is reused. This approach can be used together with different hardening techniques, since it does not need code modifications and does not introduce additional performance penalties.

3.1. CPU Checker Architecture

In this subsection, the architecture of the proposed hardware module, named *CPU Checker*, is described. The *CPU Checker* is in charge of detecting the occurrence of faults by means of checking the execution of each task which is replicated on different instants of time or on different CPUs. This module contains the following sub-modules: a memory unit, the *Checker's Controller* and one or more *CPU Observers*. Figure 2 shows the proposed architecture; detailed information about each component of the *CPU Checker* are given next:

- *Memory unit*: this sub-module is in charge of storing all the information needed to detect the occurrence of a fault. In order to program this memory block, the CPU Checker can be connected to the system bus as a peripheral, or externally accessed. This memory is divided in four different blocks to store the following information: the first and last instruction addresses of the critical task to be checked, the maximum time allowed to execute the task, and the signatures that have been calculated from the replicas of the monitored task. The number of addresses in this memory block is the number of different critical tasks that are executed on which errors need to be detected.

- *Checker's Controller*: it is in charge of managing the error detection process.
- *CPU Observers*: they observe individual executions of the replicated tasks. The number of CPU Observers in the CPU Checker depends on the number of replicas of critical tasks that can be executed simultaneously.

Each CPU Observer is composed of the following three main elements:

- The *MISR module* computes a single signature for the execution of the replica of the critical task using data coming from the trace interface of a processor.
- The *Watchdog Timer* checks whether or not the task finishes within the allowed time.
- The *Observer's Controller* manages the signature generation and Watchdog operation.

When the number of replicas of a task is more than two, the system is not only capable of detecting the occurrence of faults, but is also capable of identifying the processor where it occurred, and contribute in error correction schemes.

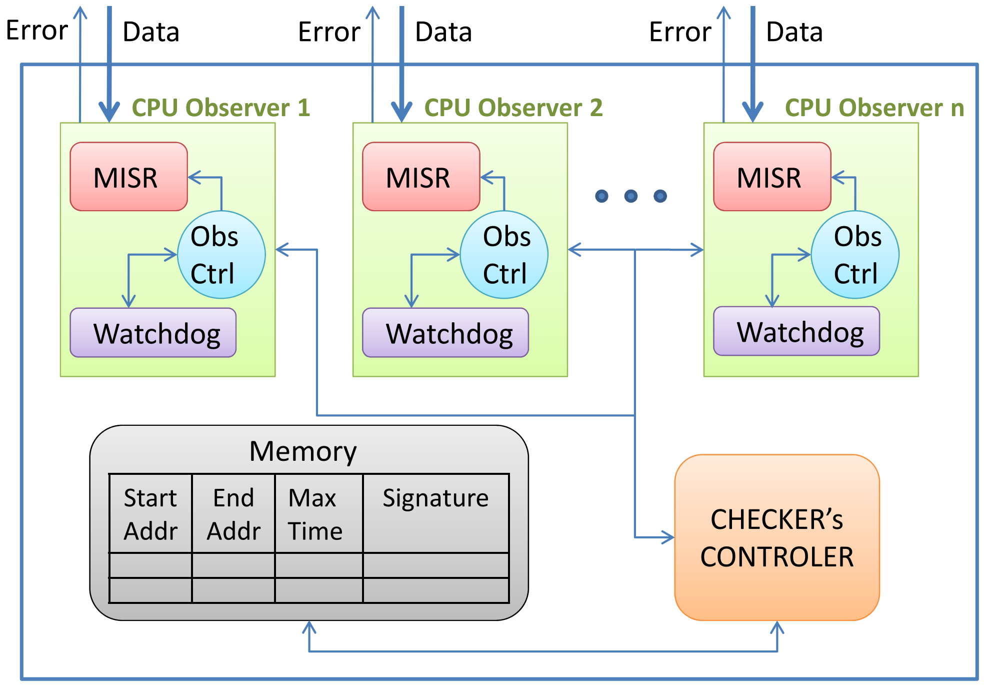

Figure 2. CPU Checker's architecture

After presenting the architecture of the *CPU Checker*, it is necessary to explain how this module works:

- The first step is to configure the system by storing in the memory block the first and last instruction addresses for each task that needs to be monitored as well as its maximum execution time and, whenever needed, activate instruction tracing.
- Once the system is initially configured, the various tasks start executing on the processors in due time. Whenever a task that needs to be checked starts its execution, a *MISR module* starts to compute a signature for this task.
- When the execution of a task has finished, the calculated signature is also stored in the memory block.
- After all the replicas of a given task have finished their execution, the computed signatures are compared. If they differ, a fault is detected and a warning signal is raised. There are three main causes that can produce an error detection:
    - The instructions executed by a task and its replicated version are different (signatures differ).

        - The expected order for the different replicas execution is altered (for the systems where this condition is relevant).
        - One of the replicas does not finish its execution (time-out).
- If no error was detected, then the processors continue to execute other tasks. Otherwise, the processors will carry out the appropriate recovery actions.

When implementing a CPU Checker in a system, it has to be noticed that some factors can have a strong influence on the final signature computed by an MISR module. One of them is the type of information we use to compute the signature (for example, the complete trace bus or only a subset of signals) and another is how often this information is sampled. With respect to the first parameter, some experimental results are presented in Section 5 in order to analyze how the type of observed information affects the error detection capability of the proposed technique. Regarding the sampling frequency, we consider two main options taking into account the debugger features present in modern microprocessors. These two options differ on how the data is introduced in the MISR module. In the first option the data introduced in the MISR module is in a cycle by cycle basis, i.e. sampling data that appears on the trace bus at every clock cycle. This case is suitable when the trace bus is directly accessible. Whether this data changes or not is not relevant for the signature generation. In the second option, the signature is obtained by compacting the data stored in the trace buffer, and then data are sampled only after some condition is verified, in order to prevent the buffer overflows. Section 5 presents the results of the experiments performed to analyze the influence of this factor on the fault detection capability of the proposed solution.

3.3. Applicability

In the previous sections we have seen the architecture of the *CPU Checker* and how it works. In this subsection the applicability of the *CPU Checker* is studied. The *CPU Checker* can be used with various system architectures combined with different hardening techniques; in any case, no extra code is added to the critical tasks nor any change is introduced in their execution. In the following paragraphs, we provide further information on how the module we present works with techniques such as time redundancy, hardware redundancy and in the case of multi-core systems.

<u>Time redundancy</u>

Time redundancy is a hardening technique based on repeating the execution of a critical task at different instants of time. The method we propose checks these repeated executions to detect transient fault occurrences by means of information coming from the debugging trace. Also, there is no need to use specific software to detect the occurrence of faults since this is done via the *CPU Checker*. When an error is detected by the module, it warns the processor about its occurrence and different recovery actions can be carried out (e.g. repeating the execution of the task, activating an alarm, etc.). With this technique the fault can be not only detected but also masked if the execution of the task is triplicated instead of duplicated.

In this case, there is only one CPU Observer in the CPU Checker since only one task is executed. Permanent fault detection can be achieved in this case with the periodic execution of a test-devoted task whose outcome is known a priori.

<u>Hardware redundancy</u>

Another possibility to make a system fault tolerant is by repeating the execution of a critical task in two or more identical processors. This technique is called *Hardware Redundancy*. With the proposed method, the *CPU Checker* checks the execution of the repeated tasks in order to check the occurrence of faults. If a fault is detected, the *CPU Checker* warns the involved processors. The advantage of this method is that it represents a non intrusive solution; it does not alter the microprocessor's normal operation and no extra computational effort is required.

When working with hardware redundancy, several operation modes can be found. One of these modes of operation is the lockstep mode. In this mode, the same process runs on different processors and a comparison between the produced signals is performed on a cycle-by-cycle basis; in this way, the fault latency achieved is minimal, whereas the amount of resources needed is higher. The method we propose works independently of the used operation mode, i.e., there is no need to modify the CPU Checker when working with any of these operation modes.

As in the previous case, if we use three processors instead of two (i.e., if we use TMR) single faults can be masked. By using TMR it is also possible to detect the occurrence of a permanent fault affecting one of the cores and to perform recovery actions like replacing or removing the faulty module.

Multiprocessor environment

In a multiprocessor environment maximum efficiency can be achieved since various processors execute different tasks at the same time. In this kind of architectures, on-line error detection can be implemented if both time and hardware redundancy are applied profiting from the available cores. In this type of environment, some cores can be used to add redundancy to the critical tasks, i.e., two or more cores execute the same task and the results are compared. The management of the task's execution can be done by different mechanisms (e.g., lockstepped pipelines or redundant multithreading [18]).

If we use the CPU Checker to detect the occurrence of faults, the execution of the various tasks is not altered when using the proposed method. Because of this fact, it is possible to use the module we propose with several redundant mechanisms. The *CPU Checker* obtains the information from the trace interface of each processor and then checks for the occurrence of faults. In case an error has occurred the module warns all the processors involved in the execution of the critical task.

4. Case studies: analysis of implementation issues

The proposed methodologies can be applied to systems where processors are used, which are equipped with debug infrastructures and interfaces that make the processor trace information available, either on-line or through buffering. To prove the feasibility and the effectiveness of the proposed techniques, two microprocessor architectures have been studied, namely LEON3 and ARM7TDMI. In the former case, the CPU Checkers have been integrated in the digital part of a SoC design. In the latter, the module was connected to the available external trace port. For each of them, an analysis of the available integrated debug interfaces has been conducted, and suitable CPU Checkers have been developed. The following paragraphs describe the studied microprocessor architectures and outline the corresponding implementation costs.

4.1 LEON3

LEON3 [30] is an open-source synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 RISC architecture, featuring a 7-stage pipeline and based on the AMBA 2.0 AHB/APB bus system. The LEON3 library includes a series of IP cores that make the model highly configurable, and can be used in systems including up to 16 processing cores attached to the same bus. Experiments have been performed resorting to different configurations with different numbers of cores.

LEON3 core has a circular instruction trace buffer that stores executed instructions. The trace buffer parallelism is 128 bits, while its dimension can be set in modules of 1 KB (a 128-entry buffer thus needs 2 KB). Many of current microprocessors have a trace bus directly accessible and in order to prove the feasibility of our approach we have modified the LEON3 interface by making the trace bus accessible from the outside of the core. Hence, the *CPU Checker* can read trace data to detect a possible fault. Nevertheless, the proposed methodology can be applied reading data from the trace buffer, at the expense of additional latency and reduced error detection coverage.

In Section 3 we described the general architecture of the proposed *CPU Checker* and how this architecture can be used for different hardening techniques. In this section, we describe how to connect the *CPU Checker* to as many LEON3 processor cores as the used system architecture needs, and we analyze the area overhead introduced by the system.

The area overhead does not only vary with the used processor, it does also vary because the size of *CPU Checker* depends on the used system architecture. The size of the *CPU Checker* varies with the number of tasks to be checked and with the number of replicas executed simultaneously, since the size of the memory and the number of *CPU Observers* depend on these two variables, respectively. For the area overhead analysis, we considered a fixed number of tasks, although the number of replicas per task may differ.

When time redundancy is used to harden a system we need the *CPU Checker* and the LEON3 processor with only one core. In this case, to implement the hardening technique only the *CPU Checker* is needed (i.e., no extra hardware), since the microprocessor is also needed for the original system.

In case of using hardware redundancy to harden a system, extra hardware is needed apart from the *CPU Checker*. Thus, for implementing this hardening technique the area overhead is the corresponding to the *CPU Checker* plus the number of extra cores. The size of the *CPU Checker* would depend on the number of redundant cores.

If a multiprocessor environment has to be hardened, then only the *CPU Checker* is needed, since the original system already includes various cores, and we just profit from one or more available cores to replicate the execution of a critical task. In this case, the size of the *CPU Checker* does not vary necessarily with the number of processors, since not every processor has to run a replica of a task. This size depends on how many times a critical task is replicated (this fixes the number of signatures to store) and how many tasks are executed in parallel (this determines the number of necessary *CPU Observers*).

Some experiments have been carried out in order to analyze in a practical manner the area overhead introduced when using the following system architectures: time redundancy with duplication of tasks, hardware redundancy with two identical processors, and multiprocessor environment based on dual core architecture. These experiments have been performed running an application consisting of a multiplication of two 16x16 integer matrices, which was split in five different critical tasks. To carry out the experiments the *CPU Checker* was configured so that it could handle the five critical tasks, i.e., the size of the memory had to be enough to store information of these tasks, and with one *CPU Observer* for the Time Redundancy case and two *CPU Observers* for the Hardware Redundancy and Multi-core cases.

Table I shows the area results obtained with ISE for these experiments. The hardware was mapped on a Virtex5 FPGA (XC5VLX110T) from Xilinx and the area results are shown in terms of LUTs, FF and memory blocks. Figure 3 graphically represents the results obtained for the carried out experiment with the various system architectures. This figure shows that the area occupied by the *CPU Checker* is minimal with respect to the total area of the original system. The overhead introduced in the hardened system is also minimal for the case of Time Redundancy and Multi-core environment. In the case of Hardware Redundancy more area is needed to harden the system, although most of it belongs to the core replicas needed (named extra hardware in Table I). The size of the CPU Checker is smaller in Time Redundancy since it contains only one CPU Observer because only one task is executed.

TABLE I. AREA OCCUPIED FOR LEON3

		Time Redundancy			**Hardware Redundancy**			**Multi-core environment**		
		LUTs	**FF**	**Memory Blocks**	**LUTs**	**FF**	**Memory Blocks**	**LUTs**	**FF**	**Memory Blocks**
Original Leon3 System		6,793	3,261	58	6,793	3,261	58	12,485	5,799	72
Hardened System	Extra HW	0	0	0	5,692	2,538	14	0	0	0
	CPU Checker	356	348	0	467	416	0	467	416	0
Overhead		5.24%	10.67%	-	90.66%	90.56%	24.13%	3.74%	7.17%	-

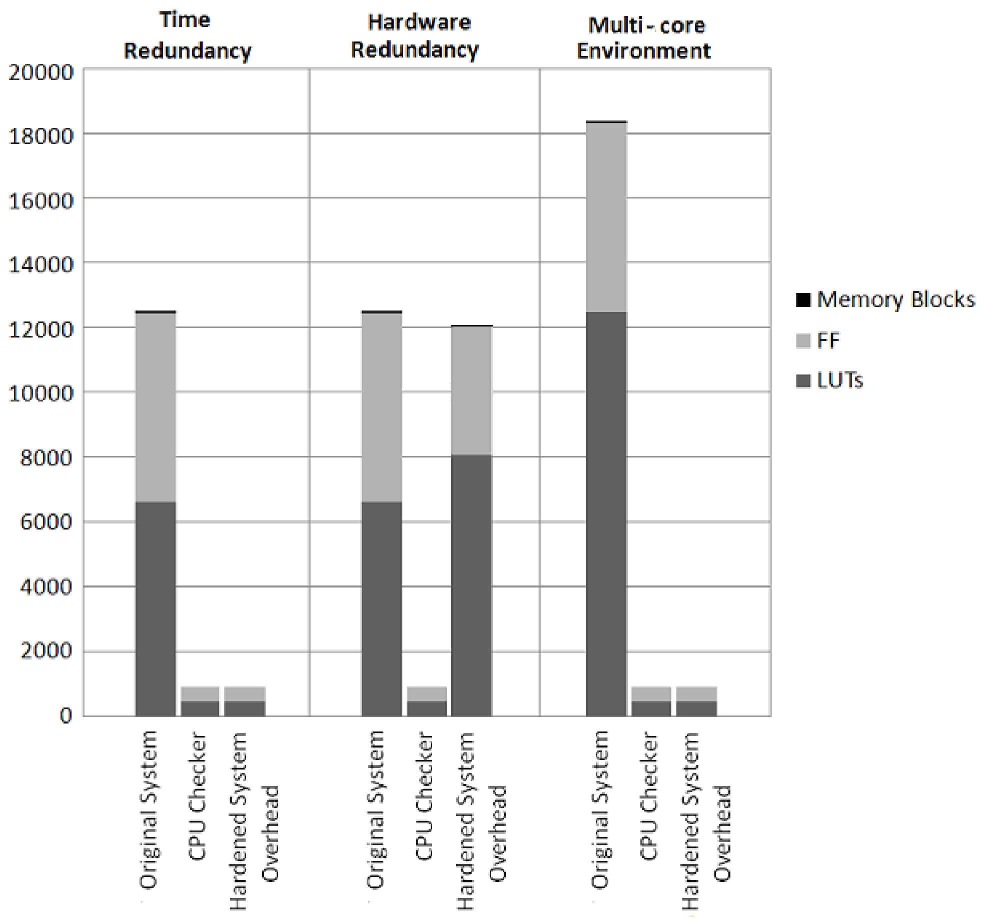

Figure 3. Occupied area

Latency

In this section the latency of the hardened system is analyzed. This latency depends mainly on two factors: the time that the critical tasks take to complete their execution (execution time) and the used system architecture. Firstly, we analyze how the latency changes with respect to the execution time of tasks, and then how it varies with the used system architecture.

The latency depends on the time that the critical tasks need to complete their execution, which will be proportional to the size of the critical tasks to be observed. First of all, we need to consider that the CPU Checker waits by far until the signature is calculated and then until the execution of the task has finished. In order to analyze the latency with respect to the size of the tasks and to provide an easier understanding, we make the assumption that the tasks are duplicated and they run one after the other (the worst case for latency). On the one hand, we can have a task with size equal to one instruction (minimum possible size); in this case the latency of the system will be the minimum possible since at every instruction the signature will be compared and the fault could be detected. However, the size of the CPU Checker is larger with this configuration, since many critical tasks are necessary to test a critical application. On the other hand, we can have a critical task that includes all the instructions of the critical application; in this case, the latency will be, at the most, two times the number of cycles that are needed to execute all the instructions. However, the size of the CPU Checker is minimal, since there is only one critical task to be checked. The relation between the area and the latency is linear for this configuration. Thus, we can state that the more times a critical application is partitioned (i.e., more tasks with fewer instructions per task), the latency will be smaller. In order to obtain the best results a trade-off between the area needed and the latency should be taken.

The other factor that affects the latency is the used system architecture. If Time Redundancy is applied to harden a system, then one task is executed and afterwards this task is executed once again, so the latency is at the most two times the number of cycles to finish the critical process. When Hardware Redundancy or Multi-core environments are used, the latency will depend on the operation mode. When working in lockstep mode, all the replicated tasks execute at the same time, so the latency is the time it takes to finish the execution of the task. If other mode of operation is used, the latency will depend on the difference between the beginnings of execution of the replicated tasks (the bigger the difference, the bigger the latency).

4.2 ARM7TDMI

The ARM7TDMI-S [31] is a general purpose 32-bit microprocessor based on RISC (Reduced Instruction Set Computer) principles. It has a Von Neumann architecture with a single 32-bit data bus and a three-stage pipeline. An Embedded Trace Macrocell (ETM) can be connected to the microprocessor core (not to the main bus) in order to perform real-time tracing. Some experiments have been performed on an LPC2129 from NXP [32] that is a microcontroller based on an ARM7TDMI with an ETM (version 1.2) connected to it. Thus, in this case, the trace bus is not directly accessible (like in LEON3), but the traced data are stored in a trace buffer accessible through the trace port.

The ETM [33] provides instruction and data trace that is stored on a FIFO buffer. Figure 4 shows the block diagram of the ETM.

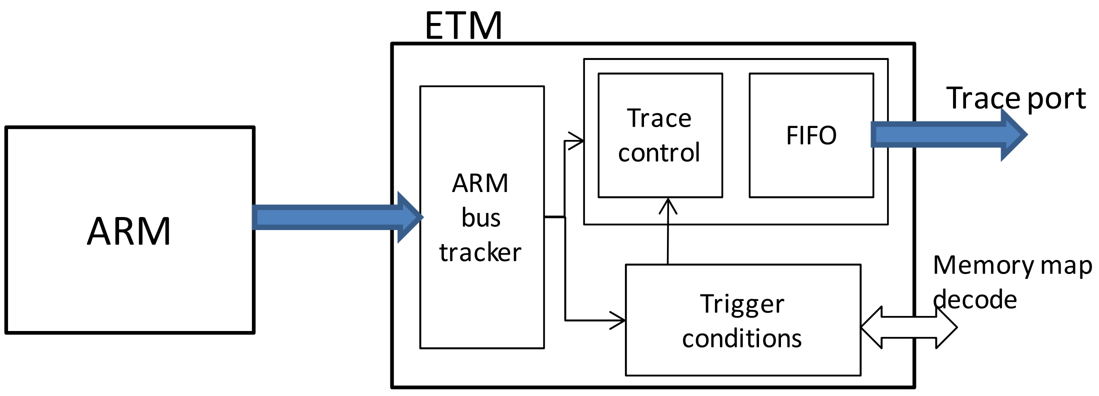

Figure 4. Block diagram of ETM

With respect to instruction trace, the program counter can be observed. In order to compress the data to be traced, only branch addresses are output. Regarding data trace, address data and/or value data can be sampled. Tracing is controlled by selecting trigger conditions and filtering signals to minimize the data to be traced (for instruction as well as for data tracing). By reducing the data to be traced, the necessary bandwidth through the trace port is reduced. Trace port size can vary from different versions and implementations of the ETM. The ETM integrated in the LPC2129 used in the performed experiments consists of 9-pins:

- 2 signals for synchronization (*traceclk*, *tracesync*)
- 3-bit signal (*tracepipe*) for indicating what is happening in the execute stage of the processor pipeline cycle by cycle
- 4-bit signal to output traced data (*tracepkt*). The size of this signal can be up to 16 bits depending on the implementation of the ETM.

When the FIFO buffer overflows, different actions can occur also depending on the version and implementation of the integrated ETM. In the used LPC2129, tracing is suspended until the buffer is read. Later versions provide more advanced options. In fact, with the newest versions, an increasing number of capabilities and enhanced features are available, which also facilitate and improve their utilization for error detection tasks.

The general architecture of the proposed *CPU Checker* is also applicable to this case study but additional registers are necessary for recovering the traced data through the narrow trace port. Thus, an additional 32-bit register (word size) is required in each *CPU Observer*. Furthermore, the functionality of the *Observer's Controller* has to include the control of the register operation. On the other hand, ETM facilities have to be configured by means of a JTAG interface. Therefore, a module in charge of managing the JTAG interface to configure the ETM registers has been developed. TABLE II. shows the logic resources necessary to implement the corresponding CPU Checker and the ETM configuration module in the XC5VLX110T FPGA.

TABLE II. AREA OCCUPIED BY ARM7TDMI

	LUTs	**FF**	**Memory Blocks**
CPU Checker	484	450	0
ETM configuration block	257	131	0

The ARM7TDMI used is a hard core and the number of logic resources required for its implementation is not available. Results show the additional hardware needed to implement the error detection tasks are in the same order than for LEON3 case. The analysis of the area overhead involved by the different hardening techniques follows the same trend as in LEON3 because it mainly depends on the redundancy technique used and just marginally on the processor used.

Latency

The latency results are slightly different from those obtained for LEON3 since in the case of ARM7TDMI traced data are read through a narrower trace port, which decreases the process speed. Furthermore, data compression techniques have influence in the time interval between consecutive data readings. For example, if a fault affects the execution flow, the latency for the ARM case is higher, since only branch addresses are sampled.

5. Fault detection capability assessment

Fault injection experiments have been performed on the available HDL description of a system-on-chip integrating the LEON3 core in order to evaluate the effectiveness of the proposed methodology in detecting errors caused by faults that occur during the processor normal operation. The implemented system consists in two cores in order to duplicate the execution of critical tasks. Therefore, the implemented *CPU Checker* requires two *CPU Observers*. Figure 5 shows a scheme of the implemented system. The *CPU Checker* is initially configured by one of the microprocessors through the main bus.

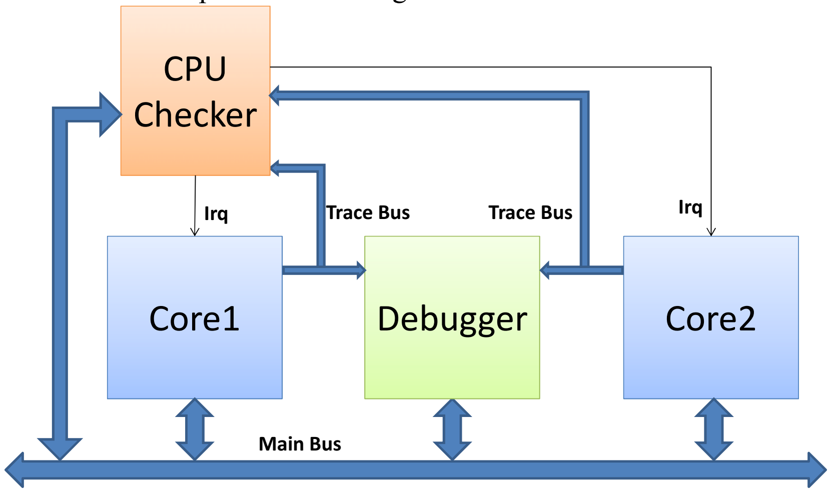

Figure 5. *CPU Checker* connection in a hardware redundancy based system

Single-Event-Upset (SEU) fault injection campaigns have been performed by applying bit-flips to the LEON3 core flip-flops while running a set of benchmark applications. SEUs have been injected randomly in space and time, and the error detection conditions detected by the

CPU Checker have been checked with respect to any error found in the execution results and in the program flow. A simulation-based fault injection setup was built using Mentor Graphics ModelSim, relying on a set of suitably developed simulation scripts.

The experiments were performed running the following sample applications:

- *Fibonacci* generates the first 20 Fibonacci numbers (3,200 clock cycles)
- *Ellip_f* is a fifth-order wave digital elliptic filter (7,390 clock cycles).

The error detection abilities obtained by employing the trace debugging infrastructure have been evaluated in two cases: the former consists in directly observing the trace bus interface exiting from the core, which is then connected to the trace buffer. The latter involves observing the data actually stored into the buffer. Since the trace interface is not sampled at each clock cycle, but just once for each instruction (or cycle in the case of multi-cycle instructions), the first method provides a richer set of information to the checker.

Table III and IV present the results for SEU fault injection experiments performed with the *Fibonacci* and the *Ellip_f* benchmarks, respectively. In all cases 10,000 random faults are injected, first on the complete core, then just on the integer unit. Experimental results are reported considering the observation of all trace fields (opcode, load/store parameters and program counter), just considering the opcode field, and considering opcode and load/store parameters. Different cases can be distinguished: *silent* faults are the ones which do not cause any error on the elaboration results or on the program flow, and are not detected by observing the processor trace; *detected* faults are those correctly identified by the proposed methodology; *false* detected faults depend on faults which do not cause any error on the results or program flow, but have an effect on the processor trace and, being detected, may add some system performance penalty; finally, *undetected* faults are the ones that, even if they cause a misbehavior, cannot be identified by the proposed approach.

With the Fibonacci benchmark, 10.88% of the faults injected in the core cause an error in the results and/or on the program flow, while 12.59% of the ones injected in the integer unit provoke errors. In the Ellip_f case, errors are caused by 10.40% of the faults injected in the core, while 11.91% of the ones injected in the integer unit provoke errors.

TABLE III. FAULT INJECTION RESULTS FOR THE FIBONACCI BENCHMARK

		All trace observation [%]				**Opcode trace – bits (31:0) [%]**				**Load/store + Opcode bits (96:64)(31:0) [%]**			
		silent	*det.*	*false*	*undet.*	*silent*	*det.*	*false*	*undet.*	*silent*	*det.*	*false*	*undet.*
Interface	***core***	64.63	10.83	24.99	0.05	80.18	10.68	8.94	0.2	73.19	10.83	15.93	0.05
	IU	51.84	12.59	35.55	0.00	74.43	12.41	12.97	0.18	64.30	12.58	23.11	0.01
Buffer	***core***	69.80	9.75	19.32	1.13	81.91	8.67	7.21	2.21	76.20	9.64	12.92	1.24
	IU	59.33	12.45	28.07	0.14	76.92	10.98	10.49	1.61	68.64	12.29	18.76	0.30

TABLE IV. FAULT INJECTION RESULTS FOR THE ELLIP_F BENCHMARK

		All trace observation [%]				**Opcode trace – bits (31:0) [%]**				**Load/store + Opcode bits (96:64)(31:0) [%]**			
		silent	*det.*	*false*	*undet.*	*silent*	*det.*	*false*	*undet.*	*silent*	*det.*	*false*	*undet.*
Interface	***core***	65.43	10.39	24.17	0.01	80.80	10.09	8.80	0.31	73.96	10.39	15.64	0.01
	IU	52.99	11.91	35.09	0.00	75.33	11.53	12.75	0.38	65.40	11.91	22.69	0.00
Buffer	***core***	70.91	9.24	18.69	1.16	82.63	8.41	6.97	1.99	77.21	9.15	12.39	1.25
	IU	60.95	11.66	27.13	0.25	77.98	10.54	10.11	1.37	70.11	11.53	17.98	0.38

The obtained experimental results show that a larger number of faults is detected when reading the trace data before it reaches the trace buffer, at the expense of a higher false positive rate. When the trace bus is not directly accessible, the buffer interface imposes a bottleneck in the amount of data that can be used to calculate the signature without losing significant information.

False positive faults are due to different reasons:

- First, they may be due to the sampling of meaningless data to calculate the signature. In order to reduce this rate, instead of using the complete trace bus, a combination of the observation of opcode and load/store data can be used. This solution maintains almost optimal fault coverage and reduces the amount of false positive.

- Secondly, the trace bus in LEON3 is connected to registers belonging to the final stages of the pipeline, which means that traced data often correspond to instructions that have already completed their execution. Therefore, if a fault affects some register in the final stages which are not used by the specific instruction (e.g., the memory stage in a instruction other than load or store), the instruction may have been executed correctly, but trace data are not correct, thus causing a false positive.
- Finally, the instant of time when the fault occurs affects the number of false positives. The information in the trace bus does not change every cycle and it may contain data that was relevant at a certain instant and later on it is no longer relevant. If the fault occurs in a period of time when the affected information are not relevant, it may lead to a false positive.

The few undetected errors are due to faults affecting the paths connecting the core to the memory bus, where the results are correctly elaborated but fail to be correctly stored, or to system bus control signals. Complementary techniques (e.g., variable replication or code assertions) can be used to detect these errors.

Experimental data show that the higher the number of observed signals, the higher is the percentage of detected errors, at the expense of a higher number of false detected ones: this fact needs to be considered when defining the error detection strategy at system level, taking into account system requirements and available resources. The best results are provided when the traced information contain the *opcode* and *load/store* parameters. In this case, errors in control flow as well as in data can be detected. This is an advantage over other techniques that are focused on control flow error detection [8]-[11]. Furthermore, experimental results published in [4] prove that, by observing the trace bus, the fault detection capabilities are higher than by using alternative (and possibly more expensive) solutions based on observing the main bus [17].

Finally, it can be observed that the proposed approach is better suited to detect faults affecting the inner parts of the processor, i.e., the integer unit, since the trace interface is directly connected to them.

6. Conclusions

An on-line error detection technique aimed at microprocessor-based systems has been presented. This approach profits from available trace interfaces in current microprocessors to observe the behavior of the system during normal operation without introducing significant penalties, and without any modifications in the core design. This technique can be used with various system architectures combined with different hardening techniques.

The proposed solution consists in adding a module, named *CPU Checker*, connected to the available trace interface. Two different microprocessors with different trace interfaces and features have been studied, LEON3 and ARM7TDMI. In LEON3 (a soft core) the trace bus is directly accessible whilst in ARM7TDMI (a hard core) the trace interface available only allows the access to a buffer trace. An analysis among three different system architectures is presented in terms of logic resources and involved latency in error detection. Results show that the necessary logic resources to implement the proposed approach involve a low percentage with respect to the necessary resources for the complete system.

Fault injection experiments have been performed using the system based on LEON3 microprocessor to assess the error detection capability of the proposed approach. Two different implementations of the solution have been evaluated: one implementation with direct observation of the trace bus, and other one accessing the trace buffer. The error detection results show that for the first case the undetected number of errors is lower but the number of false detected errors increased notably with respect to the second implementation. In both cases the percentage of the errors detected by the proposed approach is higher than 89% (more than 99% when the trace bus is directly accessible). Results also prove that the proposed approach is very efficient to detect faults affecting the inner parts of the processor, i.e., the integer unit.

References

[1] International Technology Roadmap for Semiconductors, 2010 Update to 2009 Edition (www.itrs.net)

[2] ISO/FDIS 26262 "Road Vehicles – Functional Safety" Standard

[3] M. Grosso, M. Sonza Reorda, "Exploiting Embedded FPGA in On-line Software-based Test Strategies for Microprocessor Cores", IEEE International On-Line Testing Symposium, pp. 95-100, 2009

[4] M. Grosso, M. Sonza Reorda, M. Portela-Garcia, M. Garcia-Valderas, C. Lopez-Ongil, L. Entrena, "An on-line fault detection technique based on embedded debug features", IEEE International On-Line Testing Symposium, 2010, pp. 29-34

[5] M. Gallardo-Campos, M. Portela-Garcia, C. Lopez-Ongil, L. Entrena, M. Grosso, M. Sonza Reorda, "Enhanced Observability in Microprocessor-based Systems for Permanent and Transient Fault Resilience", Conference on Design of Circuits and Integrated Systems (DCIS), 2010, pp. 240-246

[6] A. Paschalis, D. Gizopoulos, "Effective Software-Based Self-Test Strategies for On-Line Periodic Testing of Embedded Processors", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 24, N. 1, Jan. 2005, pp. 88 – 99

[7] P. Bernardi, M. Rebaudengo, M. Sonza Reorda, "Using Infrastructure IPs to support SW-based Self-Test of Processor Cores", IEEE International Workshop on Microprocessor Test and Verification, 2004, pp. 22 – 27

[8] B. Randell, "System structure for software fault tolerance", IEEE Transactions on Software Engineering, Vol.1, N. 2, Jun 1975, pp. 220 – 232

[9] A. Avienzis, "The N-version approach to fault-tolerant software", IEEE Transactions on Software Engineering, Vol. 11, N. 12, Dec. 1985, pp. 1491-1501

[10] N. F. Ghalaty, M. Fazeli, H. I. Rad, S. G. Miremadi, "Software-based Control Flow Error Detection and Correction Using Branch Triplication" IEEE International On-Line Testing Symposium, 2001, pp. 214-217

[11] R. Vemu, J. A. Abraham, "CEDA: Control-Flow Error Detection Using Assertions" IEEE Transactions on Computers, Vol. 60, Issue 9, Sept. 2011, pp. 1233-1245.

[12] M. Nicolaidis, Y. Zorian, "On-line testing for VLSI—a compendium of approaches", Springer Journal of Electronic Testing: Theory & Applications, Vol. 12, N. 1-2, 1998, pp. 7 – 20

[13] A. Mahmood and E.J. McCluskey, "Concurrent Error Detection Using Watchdog Processors—A Survey," IEEE Trans. on Computers, Vol. 37, N. 2, Feb. 1988, pp. 160-174

[14] R. Vemu, A. Jas, J.A. Abraham, S. Patil, R. Galivanche, "A low-cost concurrent error detection technique for processor control logic", ACM/IEEE Design, Automation and Test in Europe Conference and Exhibition, 2008, pp.897-902

[15] N. Karimi, M. Maniatakos, A. Jas, A., C. Tirumurti, Y. Makris, "Workload-Cognizant Concurrent Error Detection in the Scheduler of a Modern Microprocessor", IEEE Trans. on Computers, Vol. 60, N. 9, Sept. 2011, pp. 1274-1287

[16] J.R. Azambuja, A. Lapolli, L. Rosa, F. Lima Kastensmidt, "Detecting SEEs in Microprocessors Through a Non-Intrusive Hybrid Technique", IEEE Trans. on Nuclear Science, Vol. 58, N. 3, June 2011, pp. 993-1000

[17] P. Bernardi, L. Bolzani Poehls, M. Grosso, M. Sonza Reorda, "A Hybrid Approach for Detection and Correction of Transient Faults in SoCs", IEEE Trans. on Dependable and Secure Computing, Vol. 7, N. 4, Oct.-Dec. 2010, pp. 439-445

[18] D. Gizopoulos, M. Psarakis, S. V. Adve, P. Ramachandran, S. K. S. Hari, D. Sorin, A. Meixner, A. Biswas, X. Vera, "Architectures for Online Error Detection and Recovery in Multicore Processors", IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011

[19] R. Gong, K. Dai, Z. Wang, "Transient Fault Recovery on Chip Multiprocessor based on Dual Core Redundancy and Context Saving", IEEE International Conference per Young Computer Scientists, 2008, pp. 148-153

[20] R. Hyman, K. Bhattacharya, N. Ranganathan, "A Strategy for Soft Error Reduction in Multi Core Designs", IEEE International Symposium on Circuits and Systems, 2009, pp. 2217-2220

[21] M. Abramovici, "In-System Silicon Validation and Debug", IEEE Design & Test of Computers, Vol. 25, N. 3, May-June 2008, pp.216-223

[22] B. Vermeulen, K. Goossens, "Interactive Debug of SoCs with Multiple Clocks", IEEE Design and Test of Computers, Vol. 28, N. 3, May-June 2011, pp. 44-51

[23] H.F. Ko, A.B. Kinsman, N. Nicolici, “Design-for-Debug Architecture for Distributed Embedded Logic Analysis” IEEE Trans. on Very Large Scale Integration (VLSI) Systems, Vol. 19, N. 8, Aug. 2011, pp.1380-1393

[24] S.-B. Park, S. Mitra, “IFRA: Instruction Footprint Recording and Analysis for post-silicon bug localization in processors”, ACM/IEEE Design Automation Conference, 2008, pp.373-378

[25] IEEE-ISTO 5001-2003, “The Nexus 5001 Forum™ Standard for a Global Embedded Processor Debug Interface”, Version 2.0, 2003

[26] "Embedded Trace Macrocell, ETMv1.0 to ETMv3.4, Architecture Specification", ARM Limited, 2007

[27] Xilinx MicroBlaze™ Trace Core (XMTC) (v1.00c), Xilinx, 2009

[28] A. V. Fidalgo, M. G. Gericota, G. R. Alves, J. M. Ferreira, “Real-time fault injection using enhanced on-chip debug infrastructures”, Journal of Microprocessors and Microsystems, Vol. 35, N. 4, June 2011, pp. 441-452

[29] M. Portela-Garcia, C. Lopez-Ongil, M. Garcia-Valderas, L. Entrena, "Fault Injection in Modern Microprocessors Using On-Chip Debugging Infrastructures”, IEEE Transactions on Dependable and Secure Computing, Vol. 8, N. 2, Jan. 2011, pp. 308-314

[30] www.gaisler.com

[31] “ARM7TDMI-S. Technical Reference Manual”, Rev 4, ARM, 2001

[32] “LPC2119/2129/2194/2292/2294 User Manual”, Philips Semiconductors, 2004

[33] “Embedded Trace Macrocell. Architecture Specification”, ARM, 2007