On-Chip Watchdog to Monitor RTOS Activity in MPSoC Exposed to Noisy Environment

Christofer de Oliveira, Letícia Bolzani Poehls, Fabian Vargas
Electrical Engineering Dept., Catholic University – PUCRS
Av. Ipiranga 6681, 90619-900, Porto Alegre, Brazil
vargas@computer.org

Abstract— **The use of Real-Time Operating System (RTOS) became a mandatory condition to design safety-critical real-time embedded systems based on multicore processors. At the same time, these systems are becoming more and more sensitive to transient faults originated from a large spectrum of noisy sources such as conducted and radiated Electromagnetic Interference (EMI). Therefore, the system's reliability degrades. In this work, we present a hardware-based infrastructure intellectual property (I-IP) core able to monitor the RTOS' activity in a multicore processor system-on-chip (MPSoC). The final goal is to detect faults that corrupt the task scheduling process in embedded systems based on preemptive RTOS. The I-IP core, namely RTOS-Watchdog (RTOS-WD), was described in VHDL and is connected to the address busses between the cores and their local iCache memories. A case-study based on a MPSoC running different test programs under the control of a typical preemptive RTOS was implemented and exposed to conducted EMI. The obtained results demonstrate that the proposed approach provides higher fault coverage when compared to the native fault detection mechanisms embedded in the kernel of the RTOS.**

Keywords- Hardware-Based Approach, Insfrastructure-Intellectual Property (I-IP) Core, Real-Time Operating System, MPSoC, Electromagnetic Interference (EMI).

I. INTRODUCTION

The use of Real-Time Operating System (RTOS) became an attractive solution to design safety-critical real-time embedded systems. At the same time, we enthusiastically observe the widespread use of multicore processors in an endless list of our daily applications. However, the environment's always increasing hostility caused substantially by the ubiquitous adoption of wireless technologies represents a huge challenge for the reliability of real-time embedded systems [1]. As the major consequence, these systems are becoming more and more sensitive to transient faults originated from a large spectrum of noisy sources such as conducted and radiated Electromagnetic Interference (EMI). Therefore, the system's reliability degrades. In more detail, external noisy environment such as EMI may induce power supply disturbances (PSD), which in turn, cause transient faults on electronic systems [2,3]. The International Technology Roadmap for Semiconductor (ITRS) predicts increasing system failure rates due to this type of fault for future generation of integrated circuits (www.itrs.net). For ROTS-based embedded systems, such transient faults can affect not only the application running on system, but also the RTOS executing the application [4,5]. Affecting the RTOS, this kind of fault can generate scheduling dysfunctions that could lead to incorrect system behavior [6].

Up to now, several solutions have been proposed in order to deal with the reliability problems of real-time systems [7,8,9]. However, it is important to note that such solutions provide fault detection only for the application level and do not consider faults affecting the RTOS [6]. Typically, these techniques are focused on detecting errors (on the application level) that corrupt data manipulated by the processor and/or induce application illegal control-flow execution. Regarding faults affecting the RTOS that propagate to application tasks, about 21% of them lead to application failure [6] and then, are liable to be detected by such type of solutions. Generally, these faults tend to miss their deadlines and to produce logically incorrect output results. Referencing the work presented in [4], authors demonstrated that about 34% of the faults injected in the processor's registers led to scheduling dysfunctions. Indeed, about 44% of these dysfunctions led to system crashes, about 34% caused real-time problems and the remaining 22% generated (logically) incorrect system output results. In this scenario, classic techniques focused on detecting errors (on the application level) would be liable to detect only the last class of faults (22%). To conclude, the fault detection techniques proposed up to now represent feasible solutions, but they do not guarantee that: (a) faults affecting the RTOS activity will not produce erroneous system output; and (b) that each task respects its deadline, even though the yielded output is logically correct.

In this work, we present a hardware-based infrastructure intellectual-property (I-IP) core able to monitor the RTOS' activity in a multicore processor system-on-chip (MPSoC). The final goal is to detect faults that corrupt the task scheduling process in embedded systems based on preemptive RTOS. Examples of such faults could be those that prevent the processor from attending an interruption of higher priority, tasks that are strictly allocated to run on a given core but are running on another one, or even the execution of low-priority tasks that are passed over high-priority ones in the ready-task list maintained on-the-fly by the RTOS. The IP core, namely RTOS-Watchdog (or RTOS-WD), was described in VHDL and is connected to the address busses between the cores and their local iCache memories. The RTOS-WD is based on a configurable interface to easily fit any processor core.

In previous works [10,12], the authors presented an

978-1-4673-7897-0/15/$31.00 © 2015 IEEE

approach able to detect RTOS task scheduling faults in a single-core processor, for RTOSs based on the *Round-Robin* and *Preemptive* scheduling algorithms. It is important to highlight that the RTOS-WD represents a generic passive solution and consequently does not interfere within the execution flow of the RTOS running by the system.

A case-study based on the dual-core Plasma processor IP core running different test programs under the control of a typical preemptive RTOS was implemented. The case-study was prototyped in a Xilinx Virtex4 FPGA mounted on a dedicated platform (board plus control software). For validation, the MPSoC was exposed to conducted EMI. The obtained results demonstrate that the proposed approach provides higher fault coverage and reduced fault latency when compared to the native fault detection mechanisms embedded in the kernel of the RTOS.

The paper is organized as follows: Section II describes the proposed approach to monitor preemptive RTOSs. Section III describes the case study and the fault injection setup, as well as the results obtained during the fault injection experiment. Finally, we draw the conclusions in Section IV.

II. The Proposed Hardware-Based Aproach

Differently from the work presented in [10,12] where the RTOS-WD monitored the task's execution flow in **single**-core processor, in the present work the RTOS-WD is tailored to operate in conjunction with a **multicore** processor system-on-chip (MPSoC). Fig. 1 depicts the functional block diagram of such an approach.

The RTOS-WD is connected to the embedded system's bus in order to monitor the following information: Start, Tick and Interrupt signals as well as the RAM addresses accessed during the execution of the application code and RTOS. (The Tick is an external signal generated to synchronize the processor. Based on the Tick, the RTOS triggers the task switching process.) The RTOS-WD is composed of two main functional parts: a Scheduling Event Monitor (SEM) Block, which must be attached to each one of the processor cores, and a Control Unit (CU) that processes the information issued by all the SEM Blocks and takes a final pass-fail decision. The SEM Block is composed of two sub-blocks:

(a) Task Controller (TC), which identifies the task in execution based on the address accessed by the core during the application's execution. At every clock cycle, the TC compares the address on the bus with the addresses associated to each task. If the accessed address is related to a task, the signal Task ID receives the corresponding task's number.

(b) Function Identifier (FI), which analyses the functions executed by the RTOS kernel in order to discover the scheduling process execution order. At the end of this process, the FI decodes the event that triggered the scheduling process. This is the output of the SEM Block, as depicted in Figs. 1b and 1c. In order to identify such scheduling event, the FI analyses the addresses of functions executed by the RTOS kernel to schedule tasks. Table I lists such functions that are monitored by the RTOS-WD and that are executed by the RTOS kernel to manage the task scheduling process. The complete list of decoded scheduling events for the FI is shown in Table II.

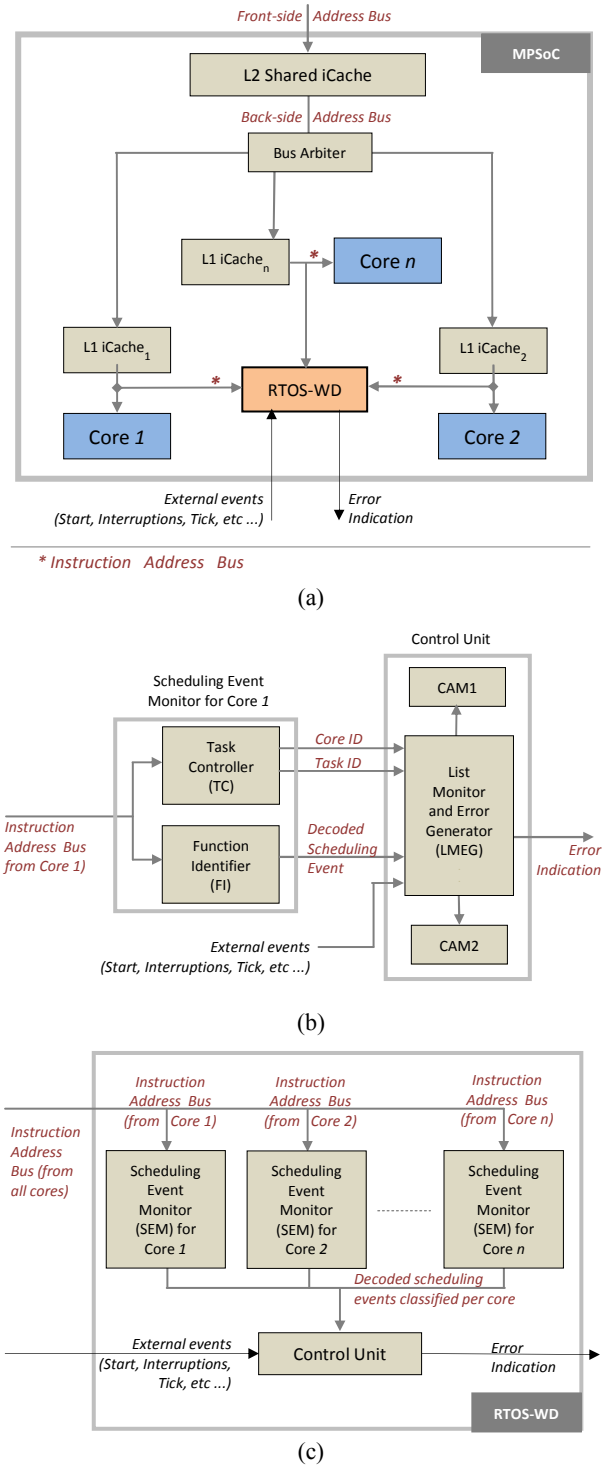

Figure 1. Proposed approach: (a) General overview at MPSoC Level; (b) RTOS-WD functional block diagram; (c) Details of RTOS-WD internal blocks[1].

[1] *Aiming to minimize the design complexity, Fig. 1c details only the SEM Block for Core 1. Nevertheless, it should be understood that there must exist a SEM Block for every core in the processor.*

The Plasma RTOS has five possible scheduling sequences (scheduling events), which are carried out by executing different combinations of the functions listed in Table I. In this scenario, the RTOS-WD must be able to recognize each one of these five sequences. Once this is done, it is enough for the RTOS –WD to follow-up the correctness by which the OS functions are called by the RTOS kernel to schedule one task to another. For instance, assume two different types of scheduling events: (a) a given task executes the system call OS_SemaphorePost(), whose goal is to unlock a semaphore and that there is a blocked task waiting for this unavailable resource to proceed computing; and (b) arrival of the synchronization signal, Tick, that triggers the context switching from the current (low-priority) task to the next ready task with higher-priority on the list of tasks ready to run; this process is initialized by the function OS_ThreadTick(). So, as observed, both scheduling events are able to trigger the context switching process, which in turn is executed by different combinations of the OS functions listed in Table I.

TABLE I
FUNCTIONS EXECUTED BY THE RTOS KERNEL AND MONITORED BY THE RTOS -WD AT THE CORE INSTRUCTIONS' ADDRESS BUS

Function (SystemCalls)	Goal
longjmp	Realize processor context switching and adjust stack pointer for the next selected task.
OS_ThreadTimeoutRemove	Remove a task from the timeout list, so that this task can be rescheduled.
OS_ThreadReschedule	Determine which is the next task to be executed, according to core, priority and state restrictions.
OS_SemaphorePost	Increment a semaphore so that to release pending tasks.
OS_SemaphorePend	Decrement a semaphore so that to block a task that is soliciting this service till it is released by another task.
OS_ThreadExit *	Indicate to the RTOS that a given task was completed and so, it should be removed from the tasks' list.
OS_InterruptServiceRoutine	This function is called to attend any kind of interruption such as external interruption or *tick*. The final goal is to call the function that will attend the arrived interruption.
OS_ThreadTick	Attend the *tick* interruption, releasing tasks by timeout.
OS_IdleThread	This is a low-priority task. It is called when there is no other ready task to run. It is also called to generate a rescheduling event, as response to a running task that released a system resource in another core.

Finally, the Control Unit is composed of a *List Monitor and Error Generator* (LMEG) Block and two *Content-Addressable Memories* (CAM1 and CAM2). The LMEG receives the *Decoded Scheduling Event* and the *Task* signals and based on this information the LMEG classifies all tasks in two separate lists, *ready tasks* and *blocked tasks*, each one organized according to their state and priority. The LMEG implements the preemptive scheduling algorithm and indicates an "Error" when a scheduling misbehavior is detected. CAM1 and CAM2 save the lists generated at runtime by the LMEG Block. The tasks labeled *ready* are stored in CAM1, while the tasks labeled as *blocked* are saved in CAM2.

To implement preemption, the algorithm with priority support keeps a list of all tasks labeled *ready* (ready-list). The tasks are sorted by their priority. Therefore, every time a *CS* takes place and a scheduling event is performed, the (*ready*) task marked with the highest priority is executed. The complexity of monitoring this kind of behavior relies on keeping track of the *ready*-list: its elements must not have any pending IO requests or semaphore objects still to be acquired. In order to accomplish this task (keeping track of the *ready*-list), the RTOS-WD should monitor not only the task addresses, but also the addresses related to the kernel synchronization, more specifically, the addresses of the functions (Systems Calls) depicted in Table I. To do so, an execution flow analysis is adopted as solution, since the function parameters remain unknown. In this solution, the RTOS-WD observes the *order* in which the functions are being called to infer the *ready*-list constraints. More specifically, the RTOS-WD analyses a control-flow graph (CFG) that represents the flow by which the functions (seen in Table I) are executed by the RTOS kernel. To illustrate this mechanism, Fig. 2 shows a situation where *Task1* is running and tries to acquire a semaphore. The system call is performed and the RTOS kernel realizes that the semaphore is already locked. In order to prevent the system from going into a deadlock as well as to increase the CPU usage, the kernel performs a *CS* calling another task into execution. The resulting execution flow for an already locked semaphore consists of: *SemaphoreLock()* and *ReSchedule()*. When the RTOS-WD detects this flow, it will infer that *Task2* is *running* and therefore is taken out from the *ready*-list (after checking the correctness of the priority policy).

TABLE II
DECODED SCHEDULING EVENTS (INPUTS TO CONTROL UNIT)

Event Description
Rescheduling due to unavailable resource
Resource release (does not generate task scheduling)
Rescheduling due to semaphore release
Event of a Tick
Rescheduling after occurring an interruption
Resource release due to time-out
End of a task
Rescheduling due to Idle Thread

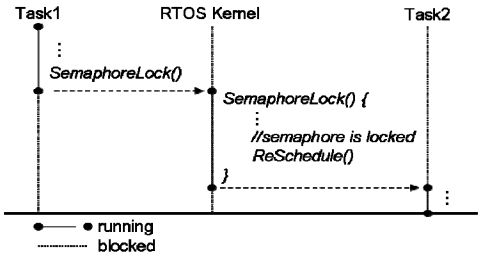

Figure 2. Example of a task schedulling flow tracked by the RTOS-WD.

A similar analysis can be performed for other situations, always concentrating all efforts in keeping the detection algorithm generic enough for any RTOS or processor. As further positive effect, this type of analysis has rendered dispensable the *Tick* signal. In more detail, the RTOS-WD detects the *Tick* by recognizing the following execution flow: *Interrupt()* and *ReSchedule()*.

A) Discussions About Fault Detection Latency

Noting that it is not possible for the RTOS-WD to monitor the parameters of system calls (only the addresses of the functions are captured by the RTOS-WD), the RTOS-WD cannot always guarrantee fault detection faster (lower fault latency) than the native fault-detection structures of the RTOS kernel. For instance, assume that a given task executes the system call *OS_SemaphorePost()*, whose goal is to *unlock* a semaphore and that there is a blocked task waiting for this unavailable resource to proceed computing. However, it happens a fault during the system call execution of *OS_SemaphorePost()* which prevents the RTOS from actually releasing the semaphore. From the point of view of the RTOS-WD, it will assume that this semaphore was released and will check for the next (ready) task to be executed. If this task is the one with the highest priority to run, then no fault is detect yet. However, the RTOS-WD will also observe that the number of blocked tasks is reduced by *one* unit by one side, and the number of unavailable resources remains *unchanged* by the other side (which is not possible since a semaphore was released from the RTOS-WD point of view). Then, at this moment the RTOS-WD will signal an Error! However, if this fault does not prevent the native fault-detection structures of the RTOS kernel from detecting that the semaphore was not properly released, they can instantly check this condition by executing a very simple and specific system call (*OS_Assert()*) which tests the status (locked or unlocked) condition of the semaphore and signal an Error!

III. Experimental Results

This section evaluates the fault detection capability of the RTOS-WD with respect to the native fault detection mechanisms of the RTOS kernel. This goal was attained by developing a MPSoC-based case study, which was exposed to conducted EMI according to the IEC 61.000-4-29 international standard [11]. In the sequence, we present the case study, the adopted fault injection approach and the obtained experimental results.

A. Case Study

To evaluate the proposed approach we adopted a case study composed of a Von Neumann 32-bit RISC Dual-Core Plasma microprocessor running a RTOS (www.opencores.org). The Plasma microprocessor is implemented in VHDL and has, with exception of the load/store instruction, an instruction set compatible to the MIPS architecture. Moreover, the Plasma's RTOS adopts the preemptive scheduling algorithm with priority support composed of the following three states: *blocked*, *ready* and *running*. The Plasma's RTOS provides a basic mechanism able to monitor the task's execution flow and manage some particular situations when faults cause misbehavior of the RTOS's essential services, such as stack overflow and timing violations. This mechanism is implemented by a function named *OS_Assert()*. Generally, when the argument of the *OS_Assert()* function is false, the RTOS sends an "error message" through the standard output.

For the fault injection experiments, we developed two test programs (TP1 and TP2) that exploit great part of the resources offered by the Plasma's RTOS (i.e., the use of message queues, semaphores and interrupts). Tasks statically assigned, restricted to a specific core (core 0 or core 1) are embedded in a dashed square whose top indicates the core hosting the task. Tasks outside these boundaries have no restrictions and can run in any core. In this case, these tasks are mapped "on-the-fly" by the RTOS to run on the first available core. TP1 and TP2 are described as follows (see Fig. 3):

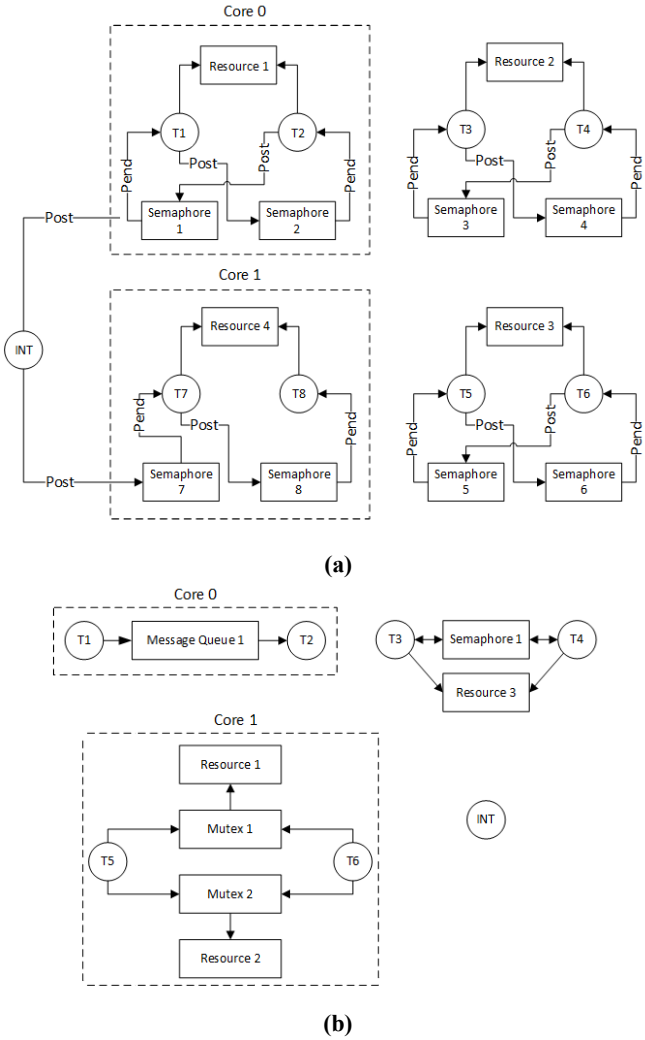

Figure 3. Functional block diagrams of the test programs: (a) TP1; (b) TP2.

TP1: 8 tasks and 1 external interrupt (*Int*) are implemented. Tasks 1 and 7 depend on the interrupt completion to start executing. Tasks 1 and 2 access and update a global variable (Resource 1), which is protected by a semaphore. First, Task 1 executes on this variable by locking (Pend) semaphore 1. When Task 1 finishes, the semaphore is unlocked (Post) and Task 2 starts executing on the same variable by locking the semaphore. Once Task 2 completes, it unlocks semaphore 2 and releases Resource 1. Tasks 7 and 8 perform similar functionality on Resource 4. Tasks 3 and 4, 5 and 6 also execute similar computation; however, they do not depend on Task Interruption to run. Tasks 1 to 8 are assigned

to the following priorities: 4, 3, 2, 1, 2, 1, 4, and 3, respectively. *Int* has the maximum priority. Tasks 1 and 2 are assigned to core 0, and 7 and 8 are assigned to core 1. Finally, Tasks 3 to 6 have no core restriction to run.

***TP2**:* 6 tasks and 1 external interrupt (*Int*) are implemented. Task 1 communicates with Task 2 through a message queue. Tasks 3 and 4 access and update a global variable (Resource 3), which is protected by semaphore 1. Further, Tasks 5 and 6 alternate accessing and updating two global variables (Resources 1 and 2), which are protected by a mutual exclusion semaphore (MUTEX). Tasks 1 to 6 are assigned to the priorities: 5, 4, 3, 2, and 1, respectively. *Int* has the maximum priority, but it remained inactive during the whole TP2 execution. Tasks 1 and 2 are assigned to core 0, and 5 and 6 to core 1, whereas Tasks 3 and 4 have no core restriction.

B. Fault Injection Setup

To perform the conducted EMI experiments, we developed a fault injection environment according to Fig. 4. In more detail, FPGA 1 is composed of the RTOS-WD and the dual-core version of the Plasma processor running the RTOS, TP1 and TP2. The Xilinx ChipScope Tool, instantiated in FPGA 2, receives two different signals: (1) the number of the *current task* in execution and (2) the *error signal*. Finally, the third component on board, named FPGA_clk, generates the clock signal to the whole system (Fig. 4a).

Fault injection campaigns were generated according to the IEC 61.000-4-29 international standard [11] by applying voltage dips to the core V_{DD} pins of FPGA 1. The nominal core V_{DD} is 1.2 volts. During the experiment, the IC peripheries remained at their regular voltage levels, i.e., 3.3 and 2.5 volts. Voltage dips were randomly injected at the V_{DD} input pins of FPGA 1 at a frequency of 25.68 kHz and consisted of dips of about 10.83% of the nominal V_{DD} (Fig. 4b). In this case, the voltage at the core V_{DD} oscillated between 1.2 and 1.07 volts. For voltage dips larger than this value, we observed the FPGA configuration loss.

C. Results' Discussion

We performed 1000 fault injection experiments per test program, totalizing 2000 experiments. It is important to point out that an experiment finishes when a fault is detected by the RTOS and/or the RTOS-WD. In this scenario, we are not able to guarantee that the observed erroneous outputs represent the total number of the faults injected during the experiments. This situation can be attributed to the fact that the adopted fault injection approach does not allow the effective control of the number of faults injected. Though, those faults that did not produce errors at the system output were considered as "*fail-silent faults*".

Figs. 5a and 6a indicate the name of the RTOS native functions (*OS_Assertion()*) responsible for the fault detection. For instance: "rtos.c:458" means that the *OS_Assertion()* function "rtos.c:458" detected 67.20% of the total number of faults detected by the kernel, during TP1 execution. As observed in Figs. 5a and 6a, the RTOS kernel detected **47.25%** of the observed faults in TP1 and **41.32%** in TP2.

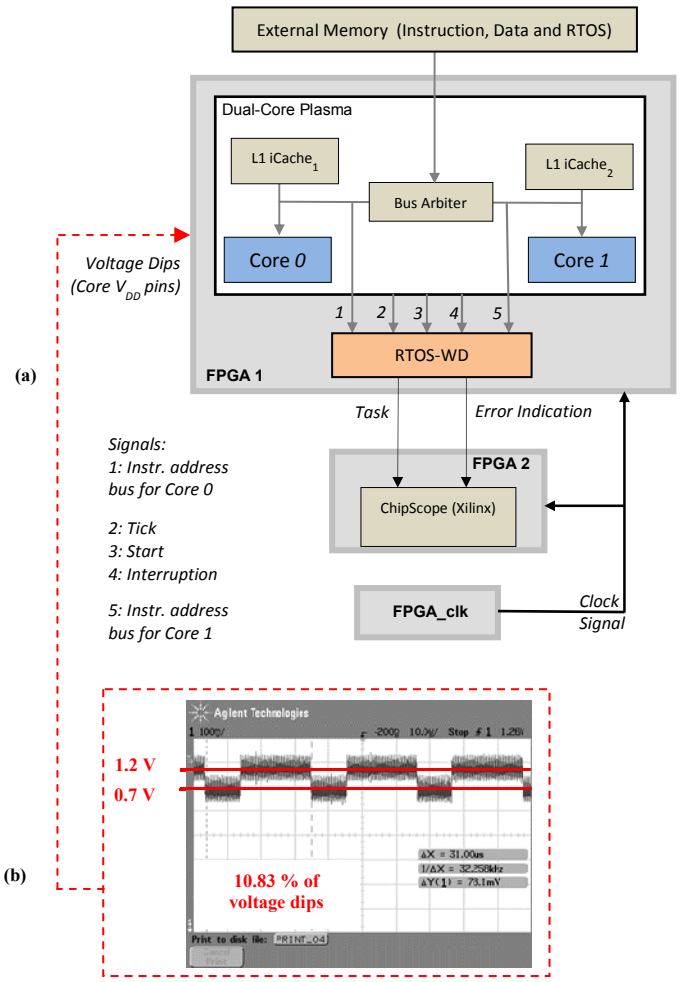

Figure 4. Fault injection environment: (a) functional block diagram of the prototyped system; (b) noise injected on FPGA1 V_{DD} pins.

Figs. 5b and 6b depict results for the RTOS-WD. The RTOS-WD was designed in such a way to identify various types of errors. Four of them occurred during the practical experiments, as observed in Figs. 5b and 6b:

- Error "**10001**": this code is generated by the RTOS-WD if the RTOS kernel does not attend an *external interrupt request* solicited by the interruption arbiter after a given period of time (defined by the designer, given in clock cycles). This is considered a software failure. 44.50% of the faults detected by the RTOS-WD were of this type when the processor was running TP1 (Fig. 5b) and 37.70% when running TP2 (Fig. 6b).
- Error "**01100**": this code is generated if the RTOS-WD observes that the interruption arbiter *did not generate the periodical interruption (tick) signal* to one or more of the cores. This is considered a hardware failure of the interruption arbiter.
- Error "**01101**": the interruption arbiter generates the *tick* signal, *but in the sequence the RTOS kernel does not call the rescheduling function to attend the event*. This is considered a software failure at the RTOS-level. 55.50% of the faults detected by the RTOS-WD were of this type when the processor was running TP1 (Fig. 5b) and 59.32% when

running TP2 (Fig. 6b).

- Error "**10110**": unexpected task rescheduling occurrence, without the occurrence of a scheduling event.

Finally, the experiments demonstrated the fault detection capability of the RTOS-WD to be 100% of the observed faults in TP1 and 97.52 % in TP2 (Figs. 5b and 6b).

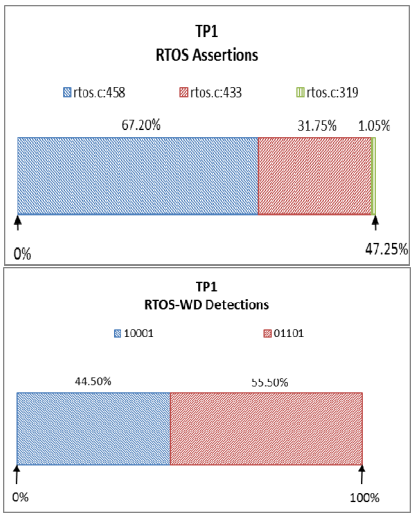

Figure 5. Experimental results for TP1: (a) Native fault-detection mechanins of the RTOS kernel; (b) RTOS-WD.

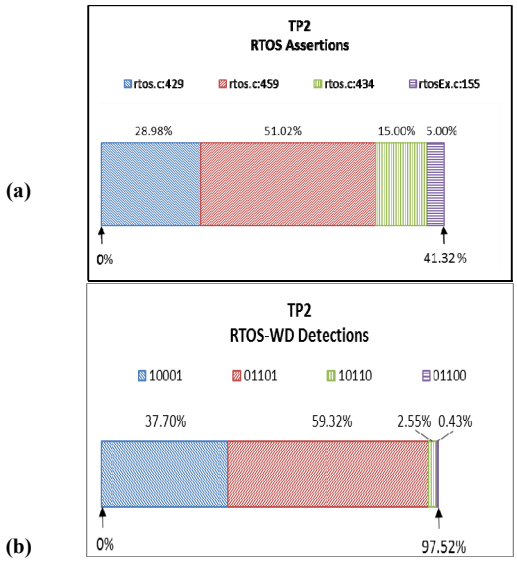

Figure 6. Experimental results for TP2: (a) Native fault-detection mechanins of the RTOS kernel; (b) RTOS-WD.

IV. Conclusions

We proposed a hardware-based approach to detect faults affecting the task scheduling process of RTOSs running in MPSoCs. It was developed an I-IP named RTOS-WD to perform on-line detection of such type of faults. It was also implemented fault injection campaigns to evaluate the effectiveness of the proposed approach. The main contribution of this paper consists of drastically improving the robustness of RTOS-based MPSoCs operating in harsh environments like those where the electronics is exposed to conducted EMI noise.

The proposed approach provides nearly 100% fault coverage, introducing only about 9.38% area overhead (8547 LUTs, Xilinx Virtex4) for a dual-core version of the Plasma processor. We assume that this number tends to further decrease when considering more complex processor architectures or MPSoCs with higher number of cores. In these cases, the corresponding area overhead shrinks because the size of the I-IP core is basically dependent on the number of tasks monitored and on the kernel complexity (i.e., number of functions existing in the RTOS kernel and how they are synchronized).

The average (TP1, TP2) fault detection capability provided by the native RTOS functions was measured 44.29%, whereas for the proposed I-IP this number raised up to 98.76%. These numbers are sustained by fault injection campaigns according to the IEC 61.000-4-29 std.

Acknowledgments

This work has been supported in part by CNPq (National Science Foundation, Brazil) under contract n. 303701/2011-0 (PQ).

References

[1] S. Ben Dia, R. Ramdani, E. Sicard, "Electromagnetic Compatibility of Integrated Circuits – Techniques for Low Emission and Susceptibility", Springer, 2006.

[2] J. Freijedo, L. Costas, J. Semião, J. J. Rodríguez-Andina, M. J. Moure, F. Vargas, I. C. Teixeira, and J. P. Teixeira, "Impact of power supply voltage variations on FPGA-based digital systems performance", Journal of Low Power Electronics, Vol. 6, pp. 339-349, Aug. 2010.

[3] J. Semião, J. Freijedo, M. Moraes, M. Mallmann, C. Antunes, J. Benfica, F. Vargas, M. Santos, I. C. Teixeira, J. J. Rodríguez-Andina, J. P. Teixeira, D. Lupi, E. Gatti, L. Garcia, F. Hernandez, "Measuring Clock-Signal Modulation Efficiency for Systems-on-Chip in Electromagnetic Interference Environment". 10th IEEE Latin American Test Workshop (LATW'09), March 2009.

[4] D. Mossé, R. Melhelm, S. Gosh, "A non-preemptive real-time scheduler with recovery from transient faults and its implementation", IEEE Trans. on Software Engineering, Vol. 29, N°. 8, pp. 752-767, August, 2003.

[5] B. Nicolescu, N. Ignat, Y. Savaria, G. Nicolescu, "Analysis of Real-Time Systems Sensitivity to Transient Faults Using MicroC Kernel," IEEE Transactions on Nuclear Science, Vol. 53, N° 4, August 2006.

[6] N. Ignat, B. Nicolescu, Y. Savari, G. Nicolescu, "Soft-Error Classification and Impact Analysis on Real-Time Operating Systems", IEEE Design, Automation and Test in Europe, 2006.

[7] S. Gosh, R. Melhem, D. Mossé, J. Sarma, "Fault-tolerant Rate Monotonic Scheduling", Journal of Real-time Systems, Vol. 15, N° 2, p. 149-181. Sept. 1998.

[8] V. Izosimov, P. Pop, P. Eles, Z. Peng, "Design optimization of time- and cost-constrained fault-tolerant distributed embedded systems", IEEE Desgin Automation and Test in Europe, pp. 864-869, 2005.

[9] Ph. Shirvani, R. Saxena. E. J. McCluskey, "Software-implemented EDAC protection against SEUs", IEEE Trans. on Reliability, Vol. 49, N° 3, Sept. 2000.

[10] D. Silva; L. Bolzani, F. Vargas, "An intellectual property core to detect task schedulling-related faults in RTOS-based embedded systems", IEEE 17th Int. On-Line Testing Symposium (IOLTS), p.19-24, 2011.

[11] "IEC 61.000-4-29: Electromagnetic compatibility (EMC) – Part 4-29: Testing and measurement techniques – Voltage dips, short interruptions and voltage variations on d.c. input power port immunity tests", First Edition, 2000-01. (www.iec.ch)

[12] C. Oliveira, J. Benfica, L. Bolzani Poehls, F. Vargas, J. Lipovetzky, A. Lutenberg, E. Gatti, F. Hermandez, A. Boyer, "Reliability analysis of an on-chip watchdog for embedded systems exposed to radiation and EMI", 9th Int. Workshop on Electromagnetic Compatibility of Integrated Circuits (EMC Compo), 2013.