IP Core to Leverage RTOS-Based Embedded Systems Reliability to Electromagnetic Interference

D. Silva[1], L.B. Poehls[1], J. Semião[2,3], I. C. Teixeira[2,4], J.P. Teixeira[2,4], M. Valdés[5], J. Freijedo[5], J.J. Rodríguez-Andina[5], F. Vargas[1]
[1]PUCRS, Brazil
[2]INESC-ID Lisboa, [3]Univ. of Algarve, [4]IST/TUL, Portugal
[5]Univ. of Vigo, Spain
paulo.teixeira@ist.utl.pt, jjrdguez@uvigo.es, vargas@computer.org

Abstract— **The use of Real-Time Operating Systems (RTOSs) became an attractive solution to simplify the design of safety-critical real-time embedded systems. Due to their stringent constraints such as battery-powered, high-speed and low-voltage operation, these systems are often subject to transient faults originated from a large spectrum of noisy sources, among them, the conducted and radiated Electromagnetic Interference (EMI). As the major consequence, the system's reliability degrades. In this paper, we present a hardware-based intellectual property (IP) core, namely RTOS-Guardian (RTOS-G) able to monitor the RTOS' execution in order to detect faults that corrupt the tasks' execution flow in embedded systems based on preemptive RTOS. Experimental results based on the Plasma microprocessor IP core running different test programs that exploit several RTOS resources have been developed. During test execution, the proposed system was exposed to conducted EMI according to the international standard IEC 61.000-4-29 for voltage dips, short interruptions and voltage transients on the power supply lines of electronic systems. The obtained results demonstrate that the proposed approach is able to provide higher fault coverage and reduced fault latency when compared to the native fault detection mechanisms embedded in the kernel of the RTOS.**

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

I. INTRODUCTION

Nowadays, several safety-critical embedded systems support real-time applications, which have to respect stringent timing constraints. In general terms, real-time systems have to provide not only logically correct results, but temporally correct results as well [1]. The high complexity of real-time systems has increased the necessity to adopt Real-Time Operating Systems (RTOSs) in order to simplify their design. Typically, these systems exploit some important facilities associated to RTOSs' native intrinsic mechanisms to manage tasks, concurrency, memory as well as interrupts. In other words, RTOSs serve as an interface between software and hardware.

At the same time, 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 [2,3]. Note that if these systems are powered by battery, the yielded reliability is even more fragile. In detail, external conditions, such as Electromagnetic Interference (EMI), Heavy-Ion Radiation (HIR) as well as Power Supply Disturbances (PSD) may cause transient faults on electronic systems [4][5][6][7]. Currently, the consequences of transient faults represent a well-known concern in microelectronic systems. The International Technology Roadmap for Semiconductor (ITRS) predicts increasing system failure rates due to this type of fault for future generation of integrated circuits [10]. In this scenario, it is worth noting that transient faults may affect not only the application running on embedded systems, but also the RTOS executing the applications. Affecting the RTOS, this kind of fault can generate scheduling dysfunctions that could lead to incorrect system behavior [1].

Up to now, several solutions have been proposed in order to deal with the reliability problems of real-time systems [11][12][13][14]. However, it is important to observe that such solutions provide fault tolerance only for the application level and do *not* consider faults affecting the RTOS that propagate to the application tasks [1]. 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 [1] and then, are liable to be detected by such type of solutions. Generally, these faults tend to miss their deadlines and to produce incorrect output results. Moreover, the work presented in [8] demonstrates 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 incorrect system output results. To conclude, the fault tolerance techniques proposed up to now represent feasible solutions, but they do *not* guarantee that each task respects its deadline.

In this paper we present a hardware-based approach to monitor the RTOS's execution flow in order to detect scheduling misbehavior. In more detail, the proposed approach provides detection of faults that can change the tasks'

execution flow in embedded system based on RTOS. In a previous work [15][16], the authors presented an Infrastructure Intellectual Property (I-IP) able to detect faults affecting the task's execution time and the task's execution flow of an embedded system running an RTOS based on the *Round-Robin* scheduling algorithm. As a further development, the present paper improves the previous work by providing the I-IP with fault detection capability for *Preemptive* RTOSs as well. Thus, a new I-IP, named RTOS-Guardian (RTOS-G), has been developed to monitor the tasks' scheduling. It is important to highlight that the RTOS-G represents a generic passive solution and consequently does not interfere with the execution flow of the RTOS embedded into the system. To evaluate the effectiveness of the proposed approach and to compare its fault detection capability against the native (software) fault detection mechanisms of the RTOS, we developed 5 different benchmarks exploiting several RTOS resources. In the sequence, we performed fault injection experiments applying conducted EMI according to the IEC 61000-4-29 international standard. Finally, the area overhead and error detection latency have been estimated.

II. BACKGROUND

RTOSs represent a key to many embedded systems and provide a software platform upon which to build applications. A RTOS is a program that schedules execution in a timely manner, manages system resources, and provides a consistent foundation for developing application code [17]. Basically, RTOSs can be classified in hard-RTOSs and soft-RTOSs. The main difference between the two categories is that a soft-RTOS can tolerate latencies and responds with decreased service quality while the hard-RTOS has to respect its deadlines, otherwise tasks' execution fails. In general terms, RTOSs provide four basic services to the application service: (1) time management, (2) interrupt handling, (3) memory management and (4) device management.

In order to optimize CPU usage, the application program is structured by the operating system as a *set of processes*. Note also that some operating systems support an additional structure level named *task*. A task can be defined as a single process or as a set of processes with data dependencies between them. Thus, tasks generally have some sort of temporal constraints on their behavior. The exact nature of these constrains depends on the scheduling model. A *deadline* is the time instant at which a process must finish its execution. The period of a task is the time interval between initiating two successive executions. Generally, a process can be in one of the following three states: *blocked*, *ready* or *running*. Further, the transfer of CPU execution from one process to another one is called Context Switch (CS).

Every RTOS has a wide range of facilities (namely, system resources), which simplify the design of real-time applications by offering native mechanisms to manage tasks, concurrency, memory, time as well as interrupts. In comparison to other (not real-time) operating systems, the efficient use of the CPU is considered the more critical and the more important issue in a RTOS. For instance, upon accessing a given embedded system resource during the execution of a task, the former mentioned mechanisms might force the task to wait for a semaphore release or some other external event before proceeding accessing the system resource. In this context, preemptive RTOSs perform a CS to force the CPU to execute another task that was labeled *ready* to run and therefore guarantee a more efficient usage of the CPU time. If there is more than one task ready to run, the decision will be made on the basis of task priorities.

Most RTOSs use scheduling algorithms based on the *Round-Robin* algorithm, which assigns equal Time Slices (TSs) to each task and executes them without priority in circular order [18]. However, in a typical real-time application, there will be tasks that must provide responses at a shorter time than others. Considering this situation, RTOSs usually implement a *Preemptive* algorithm with priority support. This results in a dynamic scheduling order and ensures time consistency for critical tasks.

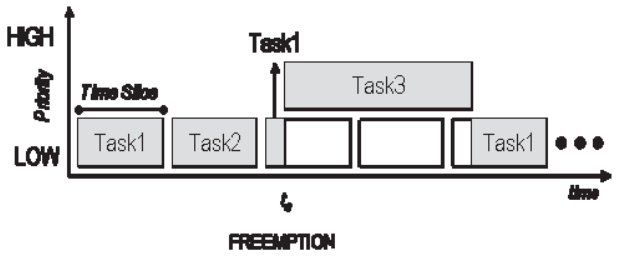


Figure 1. Preemptive scheduling algorithm

Figure 1 shows the preemptive algorithm's behavior with an example of three tasks. *Task1* and *Task2* have been given the same priority, while *Task3* has been assigned with a higher one. As specified, the TS is always the same [18] and we assume that the first two tasks do not posses external dependencies. *Task3* is blocked and waiting for an external event, but during the execution of *Task1* this dependency is solved (at time t_s) and therefore the scheduler stops the current task (*Task1*) to start executing *Task3*. Since this task has the highest priority, it will be executed completely before returning to the interrupted *Task1*. This temporary interruption of the executed task is called *preemption*. It is important to point out that preemption will take place only if a task with higher priority than the executing one is ready to run. However, if all ready tasks have the same priority, the *Tick* (a system global synchronization signal) will divide the CPU time between these tasks in equal time slices and no preemption takes place.

III. THE PROPOSED HARDWARE-BASED APROACH

This paper presents a new passive real-time scheduling monitoring approach able to detect faults affecting the RTOS running on embedded systems. The hardware-based approach has been implemented using a new I-IP, named RTOS-

Guardian (RTOS-G). Different from the version presented in [15][16], the new RTOS-G monitors the task's execution flow according to the *preemptive* algorithm. Figure 2 depicts the functional block diagram of the RTOS-G.

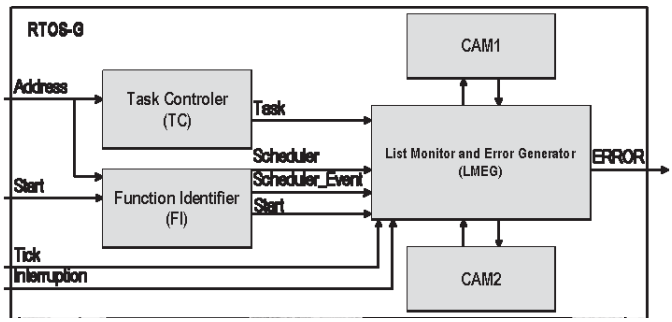


Figure 2. Functional block diagram of the RTOS-G

The RTOS-G 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. In more detail, the RTOS-G is composed of five functional blocks. The *Task Controller* (TC) identifies the task in execution based on the address accessed by the microcontroller 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* receives the corresponding task's number. The *Function Identifier* (FI) analyses the functions executed during the task scheduling process in order to check the scheduling process execution order. Finally, the FI identifies the event that triggered the scheduling process based on that order (e.g., occurrence of a *Tick* signal, IO request or semaphore acquisition). The block named *List Monitor and Error Generator* (LMEG) receives the *Scheduler_Event* signal and the *Task* in execution. 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 scheduling algorithm and indicates errors when a scheduling misbehavior is detected. As the last blocks, the two *Content-Addressable Memories* (CAM1 and CAM2) save the lists generated by the LMEG module. 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 acomplish this task (keeping track of the *ready*-list), the RTOS-G should monitor not only the task addresses, but also the addresses related to the kernel synchronization, including: *SemaphoreLock()* and *SemaphoreUnlock()*. These functions lock and unlock a previously created synchronization object which is passed by parameter to the related functions. However, it is not possible for the RTOS-G to monitor the parameters of function calls; only the addresses of the functions are captured by the RTOS-G. Consequently, the described solution does not monitor all possible fault conditions. To counteract this limitation, an execution flow analysis is adopted as solution, since the function parameters remain unknown. In this solution, the RTOS-G observes the *order* in which the functions are being called to infer the *ready*-list constraints. To illustrate this mechanism, Figure 3 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-G detects this flow, it will infer that *Task2* is *running* and therefore is taken out from the *ready*-list. 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-G detects the *Tick* by recognizing the following execution flow: *Interrupt()* and *ReSchedule()*.

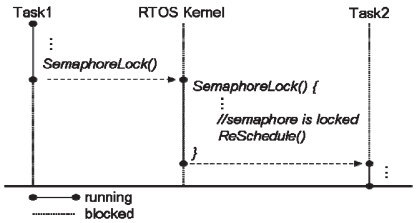


Figure 3. Execution flow analysis performed by the RTOS-G

IV. EXPERIMENTAL RESULTS

The fault detection capability of the RTOS-G with respect to the RTOS native fault detection mechanisms has been evaluated applying conducted EMI according to the IEC 61.000-4-29 international standard. In the next paragraphs, it will be presented the case study developed, the approach adopted for fault injection and a discussion related to the obtained results.

A. Case Study

To evaluate the hardware-based approach we adopted a case study composed of a Von Neumann 32-bit RISC Plasma microprocessor running an 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 *assert()*. Generally, when the argument of the *assert()* function is false, the RTOS sends an error message through the standard output.

For the fault injection experiments, we developed five different benchmarks that exploit great part of the resources offered by the Plasma's RTOS (i.e., the use of message queues, semaphores and interrupts). Figure 4 shows the block diagram associated to the five benchmarks implementing the following tasks:

- *BM1:* 8 tasks access and update the value of a global variable, which is protected by a semaphore. Indeed, another global variable is accessed by an interrupt. The 8 tasks are assigned to the following different priorities: 1, 2, 3, 4, 1, 2, 3 and 4, respectively. The interrupt has the maximum priority.

- *BM2:* 4 tasks access and update the value of a global variable, which is protected by a semaphore. The fifth task communicates with a sixth one through a message queue. Further, an *interrupt* accesses a global variable. The 6 tasks have the following priorities: 1, 2, 3, 4, 5 and 6, respectively, and the interrupt has the maximum priority.

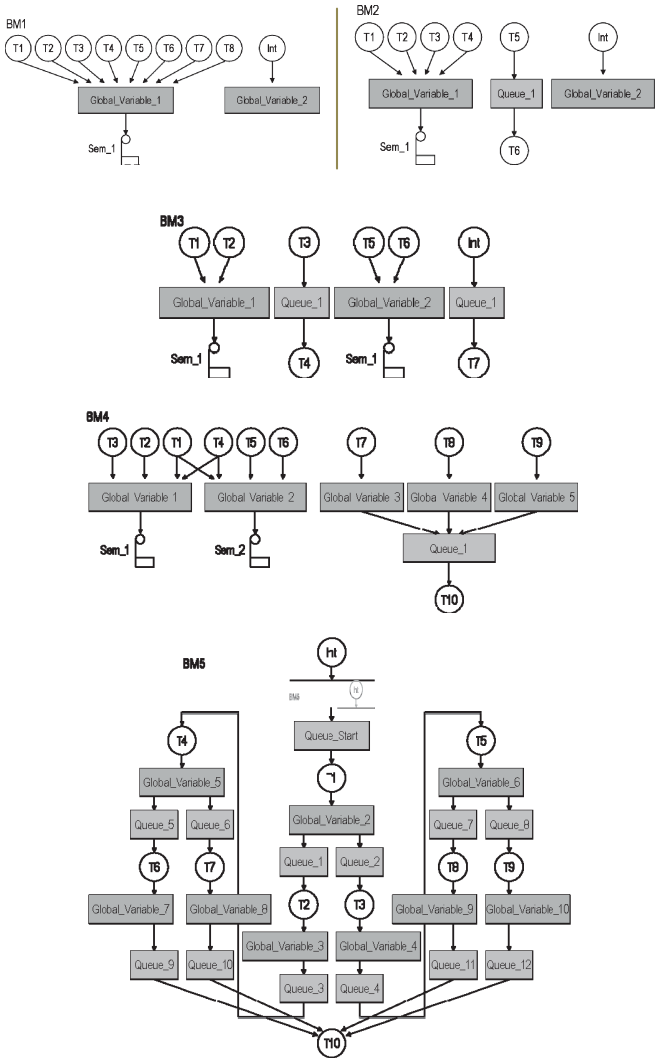


Figure 4. Functional block diagrams of the five benchmarks

- *BM3:* 2 tasks access and update the value of a global variable, which is protected by a semaphore. One task communicates with another task through a message queue. Two further tasks access and update the value of a second global variable, which is protected by a mutual exclusion semaphore (MUTEX). Finally, an interrupt communicates with a last task throughout a message queue. The 7 tasks have the following priorities: 1, 2, 3, 4, 5, 6 and 7, respectively and the interrupt has the maximum priority.
- *BM4:* 4 tasks access and update the value of *Global_Variable_1*, which is protected by a semaphore. There is also a *Global_Variable_2* that is accessed by 4 other tasks. Tasks 1 and 4 share both variables. Finally, three other tasks (T7, T8 and T9) communicate with a last one (T10) throughout a message queue. Tasks 1 to 6 have priority equal to 1, Tasks 7 to 9 have priority 2 and Task 10 holds priority 3.
- *BM5:* the last benchmark is the most complex of the five experiments and consumes the largest amount of RTOS resources. In this benchmark, an *interrupt* communicates with a task (T1) via a message queue. Then, T1 communicates with Tasks T2 and T3 via two other message queues, which in turn send messages to Tasks T4, T5, T6, T7 T8, T9 and T10, respectively, by means of queue resources as well, as depicted in figure 4. In this scenario, Task 1 has priority equal to 1, Tasks 2 and 3 equal to 2, Tasks 4 and 5, equal to 3, Tasks 6 to 9 equal to 4, and finally, Task 10 holds the highest priority: 5.

B. Fault Injection Setup

To perform the conducted EMI experiments, we developed a fault injection environment according to Figure 5. In more detail, FPGA 1 is composed of the Plasma microprocessor running the benchmarks and the RTOS-G IP that monitors the tasks' execution flow.

The ChipScope inside FPGA 2 receives two different signals: (1) the number of the *current task* in execution and (2) the *error signal*. Finally, the third block, named FPGA_clk, generates the clock signal to the whole system.

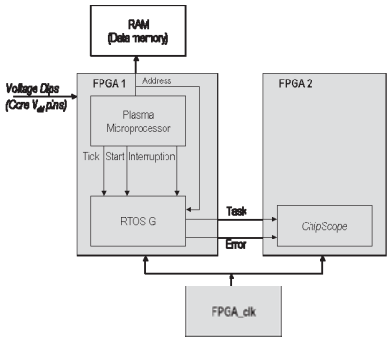


Figure 5. Fault injection environment

Fault injection campaigns were generated according to the IEC 61.000-4-29 standard by applying voltage dips to the FPGA 1 core V_{dd} pins. The nominal core V_{dd} is 1.2 volts. During the experiment, the IC peripheries remained at their normal voltage levels, i.e., 3.3 and 2.5 volts. Voltage dips were randomly injected at the FPGA 1 V_{dd} input pins at a frequency of 25.68 kHz and consisted of dips of about 10.83% of the nominal V_{dd}. For voltage dips larger than this value, we observed the lost of the FPGA configuration.

C. Results' Discussion

We performed 1000 fault injection experiments for each benchmark, totalizing 5000 experiments. It is important to point out that an experiment finishes when a fault is detected by the RTOS or the RTOS-G. 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*".

During the fault injection campaign, we classified the error´s behaviors as follows:

- *Error_1:* a blocked task is executed.
- *Error_2:* the task in execution does not appear on the monitor task's list.

- *Error_3:* the task in execution is not the one with the highest priority on the task list.
- *Error_4:* the *Tick* does not trigger the scheduling process.
- *Error_5:* the scheduling process was triggered without a *Tick* signal.
- *Error_6:* no interruption occurs upon an interrupt signal event
- *Error_7:* an interruption occurs even if no interrupt signal was produced
- *Error_8:* a scheduling event does not cause the rescheduling of the tasks in the list
- *Error_9:* the tasks of the list were rescheduled without occurrence of a scheduling event.

Figure 6 depicts the effect of the injected faults on the tasks' execution flow related to BM1 as detected by the RTOS-G. This figure indicates the relative number of errors measured when the system was executing the concerned benchmark. For example, 44.44% of the times executing BM1 the system stopped with faulty behavior during *Task1*. In this case, *Task1*, just as *Task5* with 53.55%, was assigned to the lowest priority equal to 1. It is interesting to note that during the interrupt's execution (*Int*) with the maximum priority no errors occurred at all.

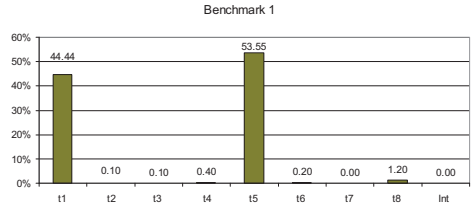


Figure 6. Tasks' behavior for BM1

Figure 7 shows the behavior of the tasks during execution of BM2 under the influence of PSD. Looking at this figure, we observe that 22.72% of the times executing BM2, an error was identified during the idle state, which is labeled with the lowest priority equal to 0.

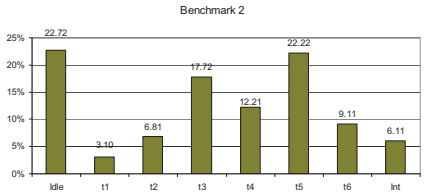


Figure 7. Tasks' behavior for BM2

The effect of faults injected into the system during execution of BM3 and the related error behavior of the tasks are observed in figure 8. As shown in this figure, 50.16% of the times executing BM3 the RTOS identified a faulty behavior during the execution of *Int*. In this benchmark, the *Int* was labeled with a maximum priority.

Figures 9a and 9b show the behavior of the tasks during execution of BM4 and BM5, respectively, under the influence of PSD. Looking at figure 9a, we observe that almost 18.00% of the times executing BM4, an error was observed during the execution of *Task 4*. Similar error occurrences were observed for the execution of *Task 1* (14%) and the idle state (13%). All of them were labeled with the lowest priority equal to 1. For figure 9b, we observe that most of the errors occurred during the execution of *Tasks 6* and *5* (almost 50% and 20%, respectively). These tasks held priorities equal to 3 and 4, respectively. It is worth noting that during the execution of *Task 10*, which held the highest priority (5), no error occurrence was observed.

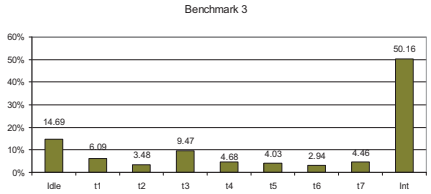
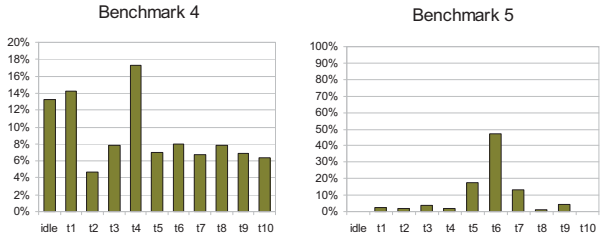


Figure 8. Tasks' behavior for BM3



(a) (b)

Figure 9. Tasks' behavior for BM4 and BM5 (error occurrence per task)

While analyzing the experimental results, we can underline the following points:

a) We observed that the use of more complex RTOS resources causes a higher error occurrence probability. In other words, the system is more likely to suffer from an error when the task in execution requires more complex resources from the RTOS. It is interesting noting that during the execution of the function "queue", the RTOS kernel executes a "semaphore" function as well. Therefore, a task using a message queue requires more complex RTOS resources than that one using only a semaphore.

b) Note that the priority level of tasks using the same RTOS resources has no direct influence on the error probability.

c) Regarding the RTOS-G fault detection capability, the results demonstrate that for all benchmarks the detection rate is nearly 100% (Table I), except for BM5 that there was 6.4% of the observed errors that were only detected by the RTOS native fault detection mechanisms. This RTOS-G detection escape can be explained by the fact that the RTOS-G, as the case for the Plasma processor, is also an embedded logic block in the FPGA. Thus, being exposed to power-supply voltage transients as well. However, it should be noted that since the RTOS-G is much smaller and build-up with a much less complex logic than the Plasma processor, it is more robust to V_{dd} transients. This reasoning may also explain the very low error detection rate of the RTOS native mechanisms (70.5%) against the one of the RTOS-G (93.6%).

d) Note also that as long as more complex resources of the RTOS kernel are used, the higher error detection rate can be yielded by the native fault detection mechanisms implemented in the Plasma's RTOS (*assert()* function). This reasoning is easily observed in Table I, moving from BM1 (2.4%) to BM5 (70.5%), and can be explained because such native fault detection mechanisms are called by the kernel every time it runs its resources (to check the proper execution of message queues and semaphores).

e) As the disturbance (insertion of voltage dips on the V_{dd} power bus) results basically on the increase of propagation delays in a uniform way through the logic circuit (FPGA), we do expect these faults to be *delay* faults. In this scenario, most of these faults induce multiple errors not only during the task dataflow execution (for

example, corrupting results provided by the Arithmetic and Logic Unit – ALU) but also during the scheduling process controlled by the RTOS (thus, manifesting themselves as *missing deadlines*). This reasoning explains the high fault detection capability yielded by the RTOS-G.

TABLE I
RESULTS OBTAINED BY FAULT INJECTION EXPERIMENTS

Benchmark	RTOS Kernel [%]	RTOS-G [%]	Faults detected only the RTOS [%]
BM1	2.4	99.9	-
BM2	25.9	100	-
BM3	45.8	100	-
BM4	60.4	100	-
BM5	70.5	93.6	6.4
Average	*41.0*	*98.7*	*6.4*

The proposed approach has also been evaluated with respect to the introduced overhead. We computed that the area overhead associated to the RTOS-G (333 LUTs) with respect to the Plasma microprocessor (3306 LUTs) is only 10%. We assume that this number tends to further decrease when considering more complex microprocessor architectures. For larger cores, one can expect the corresponding area overhead to shrink to values lower than 10% because the size of the I-IP core is basically dependent on the kernel complexity (i.e., number of resources, or functions, existing in the RTOS kernel and how they are correlated). It is also important to highlight that each of the two adopted CAMs is able to store 14 task information and accounts for 60% of the total I-IP area. In case the overhead is a critical issue, the CAMs can be placed externally. In this case, the RTOS-G's logic block introduces only 4% of overhead.

To conclude the evaluation process, it was also estimated the RTOS-G fault latency with respect to the one of the native mechanisms implemented by the Plasma's RTOS (namely, *assert()* function). The obtained results indicate that the average RTOS-G latency represents only 2% of the latency yielded by the RTOS. This time was measured by means of the ChipScope tool in terms of processor clock cycles.

V. CONCLUSIONS

In this paper we proposed a hardware-based approach to detect transient faults affecting the task's execution flow and the task's execution time of RTOS-based embedded systems. In general terms, the proposed approach targets transient faults affecting the task scheduling process of the RTOS. It was developed an I-IP named RTOS-G 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 embedded systems 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 10% area overhead. The fault detection average provided by the native RTOS functions was observed to be around 41.0%, whereas for the proposed I-IP this number raised up to 98.7%. These numbers are sustained by fault injection campaigns according to the IEC 61.000-4-29 international std. Furthermore, the introduction of the RTOS-G drastically reduces the fault detection latency to a level of only 2% of the native RTOS' latency.

Future work includes the extension of this approach to multi-processor systems on a chip (MPSoCs).

REFERENCES

[1] 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.

[2] E. Touloupis, J. A. Flint, V. A. Chouliaras, D. D. Ward, "Study of the Effects of SEU Induced Faults on a Pipeline Protected Microprocessor", IEEE TC, 2007.

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

[4] 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.

[5] 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.

[6] G. Miremadi, J. Torin, "Evaluating Processor-Behavior and Three Error-Detection Mechanisms Using Physical Fault-Injection", IEEE Transactions on Reliability, Vol. 44, N° 3, Sept. 1995.

[7] J. Arlat, Y. Crouzet, JU. Karlsson, P. Folkesson, E. Fuchs, G. H. Leber, "Comparison of Physical and Software-implemented Fault Injection Techniques", IEEE Trans. on Computer, Vol. 52, N. 9, Sept., 2003.

[8] 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.

[9] 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.

[10] http://public.itrs.net

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

[12] P. Mejia-Alvarez, D. Mossé, "A responsiveness approach for scheduling fault-recovery in real-time systems", 5th Real-Time Technology and Applications Symposium, pp. 83-93, 1999.

[13] 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.

[14] Ph. Shirvani, R. Saxena. E.J. McCluskey, "Software-implemented EDAC protection against SEUs", IEEE Trans. on Reliability, Vol. 49, N° 3, pp. 273-284, Sept. 2000.

[15] J. Tarrillo, L. Bolzani, F. Vargas, "A Hardware-Scheduler for Fault Detection in RTOS-Based Embedded Systems", IEEE 12th EUROMICRO Conference on Digital System Design, 2009.

[16] J. Tarrillo, L. Bolzani, F. Vargas, E. Gatti, F. Hernandez, L. Fraigi, "Fault-Detection Capability Analysis of a Hardware-Scheduler IP-Core in Electromagnetic Interference Environment", IEEE 7th East-West Design & Test Symposium, 2009.

[17] Q. Li, C. Yao, "Real-Time Concepts for Embedded Systems", CMP Books, San Francisco, CA, USA, 2003.

[18] A. Silberschatz, P. B. Galvin, and G. Gagne, "Operating System Concepts", John Wiley & Sons, 1997.

[19] "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).