IEICE Electronics Express, Vol.4, No.23, 755–761

HW/SW architecture for soft-error cancellation in real-time operating system

M. H. Neishaburi[a)], **M. R. Kakoee, M. Daneshtalab, and S. Safari**

School of Electrical and Computer Engineering, University of Tehran, North Kargar Ave., Tehran 14395–515, Tehran, Iran

a) *mhnisha@cad.ece.ut.ac.ir*

Abstract: Today, real-time applications with critical constraints are usually run in an environment with Real-Time Operating System (RTOS). Services provided by RTOSs are severely exposed to faults which affect both functional and timing of the tasks running on the RTOS based system. In this paper, we introduce a new architecture for RTOS provides more robust services in term of Soft Errors (SEs). We evaluate and analyze robustness of the services due to SEs in two architectures, i.e. SW-RTOS and HW/SW-RTOS. Experimental results show more robust services were provided by HW/SW-RTOS versus purely SW-RTOS regarding SEs.

Keywords: Real-Time Operating Systems (RTOS), Soft Error

Classification: Science and engineering for electronics

References

[1] Ph. Shirvani, R. Saxena, and E. J. McCluskey, "Software Implemented EDAC Protection against SEUs," *IEEE Trans. Reliab.*, vol. 49, no. 3, Sept. 2000.

[2] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Design Optimization of Time- and Cost-Constrained Fault-Tolerant Distributed Embedded Systems," *DATE*, Munich, Germany, pp. 864–869, 7–11 March 2005.

[3] S. Ghosh, R. Melhem, D. Mossé, and J. Sarma, "Fault Tolerant Rate Monotonic Scheduling," *J. Real Time Systems*, vol. 15, no. 2, Sept. 1998.

[4] N. Ignat, B. Nicolescu, Y. Savaria, and G. Nicolescu, "Soft-Error Classification and Impact Analysis on Real-Time Operating Systems," *DATE 2006*.

[5] S. Chandra, F. Regazzoni, and M. Lajolo, "Hardware/Software Partitioning of Operating Systems: a Behavioral Synthesis Approach," *GLSVLSI*, pp. 324–329, 2006.

[6] V. J. Mooney and D. M. Blough, "A Hardware-Software Real-Time Operating System Framework for SoCs," *IEEE Des. Test. Comput.*, vol. 19, no. 6, pp. 44–51, 2002.

[7] Morton and W. M. Loucks, "A HW/SW Kernel for SoC Designs," in *Proceedings of the 2004 ACM Symposium on Applied computing*, New York, USA, pp. 869–875, 2004.

[8] M. H. Neishaburi, M. R. Kakoee, M. Daneshtalab, S. Safari, and Z. Navabi, "A HW/SW Architecture to Reduce the Effects of Soft-Errors

DOI: 10.1587/elex.4.755
Received September 14, 2007
Accepted October 29, 2007
Published December 10, 2007

in Real-Time Operating System Services," *IEEE-DDECS*, 11–13 April 2007.
[9] B. Nicolescu, N. Ignat, Y. Savaria, and G. Nicolescu, "Sensitivity of Real-Time Operating Systems to Transient Faults: A Case Study for MicroC Kernel," *IEEE Radiation and its Effects on Components and Systems*, Cap de Agde, France, 19–23 Sept. 2005.
[10] Motorola HC12 CPU Awareness and True-Time Simulation, Metrowerks Corp., 2004.
[11] B. Saglam, J. Lee, and V. J. Mooney, "A System-on-a-Chip Lock Cache with Task Preemption Support," *CASES'01*, 16–17 Nov. 2001.

1 Introduction

The Real-Time Operating System (RTOS) provides an abstraction layer to hide the processor hardware details from software point of view. To do so, the RTOS kernel should provide four main types of basic services to application programs, including *Time Management* (*Timer*), *Dynamic Memory Allocation*, *Task Management* and *Inter-Process Communication* (*IPC*).

Task Management provides several services such as *task creation*, *task scheduling* and *task priority assignment*. Using these services, software developers are able to partition the design as a number of separate parts of software, called *task*, in such a pertinent way that each task handles a distinct topic, a specific goal, and maybe has its own real-time deadline. The software developers should be familiar with the Inter-Process (Inter-Task) Communication and Synchronization RTOS services to pass the information between two tasks safely. This leads to the coordination of the tasks to cooperate with each other. Due to the stringent timing requirements of most Real-Time applications, some basic timer services, i.e., *task delays* and *time-out commands* should also be provided by RTOS kernels. Additionally, Dynamic Memory Allocation allows the allocation of a block of RAM for temporary usage in application software. These blocks of memory can transfer a large amount of data between two tasks. Same kernel services are also provided by non-RTOSs. The main difference between general operating systems and RTOSs is the need for deterministic timing behavior in the RTOSs.

There are several innovations introduced by system designers to deal with the problems of Soft-Errors (SEs) in RTOSs in the literature [1, 2, 3, 4]. The method presented in [1] includes an additional application checks other applications in their workspace memory. In [2] the idea of replication of the systems is proposed, and finally in [3] the researchers were concerned with designing a robust scheduling algorithm. As shown in [4], a SE may cause a failure in the multi-tasking process of an RTOS. The faults may be propagated to the application level and thus defeat the entire fault-tolerant mechanisms and this can lead to endangering valuable assets and life. Therefore, the necessity of designing a robust fault tolerant RTOS can be concluded.

In [5], the authors suggested that by HW/SW partitioning of OSs and moving some of the RTOS functionalities, i.e. task synchronization and sched-

DOI: 10.1587/elex.4.755
Received September 14, 2007
Accepted October 29, 2007
Published December 10, 2007

uling to HW, faster executions may be obtained. They claimed that this improvement has come with the cost of only 13K gates. Although many researches have been worked on this topic [6, 7], there is still no commercial RTOS takes advantages of this feature.

The main contributions of this paper are:

1. We analyze and evaluate SE effects in services provided by purely SW-RTOS and HW/SW-RTOS.
2. We propose an effective RTOS architecture provides more effective and robust services related to soft-errors.

2 Experimental Framework

This section presents our approach for designing a HW/SW-RTOS as well as fault injection in the proposed model.

2.1 SW-RTOS

In our framework, we use *eCos* (embedded Configurable OS) as purely SW-RTOS. eCos is an open source, royalty-free and real-time operating system intended for embedded systems and applications. eCos was designed for devices with either memory footprints in tens to hundreds of kilobytes, or real-time requirements. It can be used on systems that do not have enough RAM to support embedded Linux, which currently requires a minimum of about 2 MB of RAM, not including application and service requirements.

2.2 HW/SW-RTOS

Many researchers have investigated various approaches to provide predictable and deterministic response time for RTOS at an affordable cost. One approach is to move RTOS functionality from software to dedicated hardware part; because hardware implementation of an algorithm is more predictable than software implementation of it; Moreover, it can also increase system performance due to the CPU load reduction.

The idea of a HW-OS that uses hardware implementation of Scheduling and Inter-Process communication has been proposed in [6]. In our proposed HW/SW-RTOS implementation, we replaced the POSIX support of eCos operating system with dedicated data exchanging mechanisms. The scheduler is also replaced with a dedicated hardware module. Inter-Process communications have been done by semaphores or Mutex. In our proposed HW/SW-RTOS, we directly update memory locations for implementation of semaphores and Mutex. Inter-Process communications have been done by generating standard bus transactions; consequently, they are carried out in HW/SW-RTOS more efficiently than SW-RTOS implementation.

Figure 1 shows the comparison between standard SW-RTOS (top) and our proposed HW/SW-RTOS architecture (bottom). As shown in Figure 1, in the proposed HW/SW-RTOS, the operating system is composed of five units:

DOI: 10.1587/elex.4.755
Received September 14, 2007
Accepted October 29, 2007
Published December 10, 2007

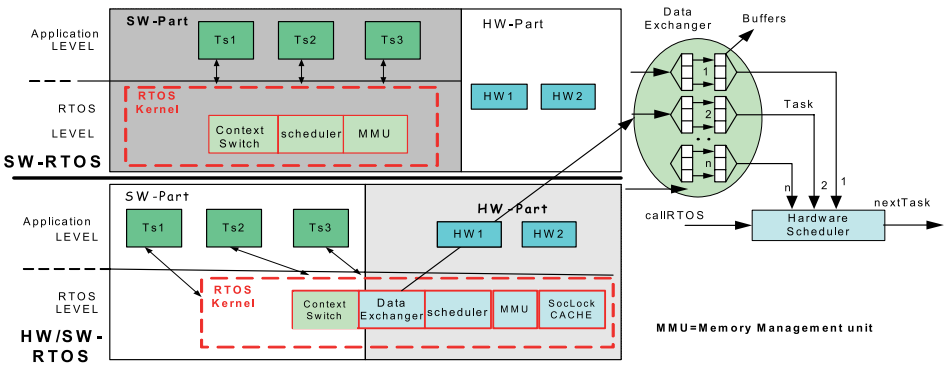


Fig. 1. SW-RTOS versus HW/SW-RTOS.

- Scheduling unit (Implemented in HW)
- Data Exchanger unit (Implemented in HW)
- System on Chip Lock Cache unit (Implemented in HW)
- Context switching unit (Implemented in SW)

2.2.1 Scheduling Unit

The scheduling unit finds out the next software task ID. In our proposed HW/SW-RTOS, the hardware implemented Weighted-Round-Robin (WRR) is used. At each scheduling cycle the task pointer increments and the *Data Exchanger* unit returns the condition of task (executable or blocking). If an executable task is found, the CPU will be informed by issuing a hardware interrupt. WRR unit selects the not-recently-executed task and avoids the starvation problem if there are several executable tasks.

2.2.2 Data Exchanger Unit

Data Exchanger manages the data transfer between tasks. If a task provides (needs) some data for (from) another task, it informs Data Exchanger unit to handle its case.

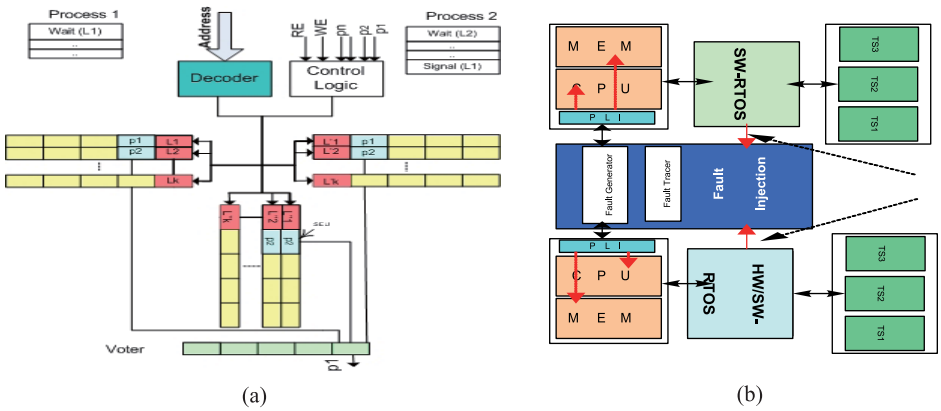
2.2.3 System on Chip Lock Cache (SoCLC) Unit

The SoCLC used in our framework to handle mutual exclusions is based on [11]. As shown in Figure 2 (a), In our method, if process p_1 executes $wait(L_1)$ instruction, it will wait until L_1 is activated. To do so, p_1 will be added to the list of waited processes for L_1. As a result of the execution of this instruction, L_1' and L_1'' will be blocked until another process executes $signal(L_1)$ instruction. However, We used Triple Module Redundancy (TMR) to keep three dedicated lists for each lock variable to mitigate the SE effects.

2.2.4 Context Switching

Typical context switching consists of three steps i.e. pushing all CPU registers to the current task stack, scheduling the next task to be run, popping all CPU registers to the next task. The steps 1 and 3 are implemented in

DOI: 10.1587/elex.4.755
Received September 14, 2007
Accepted October 29, 2007
Published December 10, 2007



(a) (b)

Fig. 2. (a) System on chip Lock Cache unit (SoCLC); (b) Fault Injection Environment.

software because all CPU registers must be stored into or restored from the memory; whereas, step 2 can be carried out in hardware by specifying the next executable task to the CPU in scheduling unit.

3 Fault Injection Environment (FIE)

A fault injection technique for SW-RTOS was proposed in [9]. However, to the best of our knowledge, there is no fault injection mechanism in the literature, which supports the fault-injection in HW/SW-RTOS except the work presented in [8]. In our method a fault can be considered as a single bit-flip in the CPU registers. Programming Language Interface (PLI) is used to access to the internal data structures of the Verilog code (e.g. CPU registers) when the application is running. As shown in Figure 2 (b), our fault injection environment consists of two main parts: *Fault Generator* decides when and where the fault will be injected and *Fault Tracer* traces the services that are currently executing in RTOS and informs Fault Generator about active services. Eventually, while the main services of the SW-RTOS (eCos kernel) and HW/SW-RTOS kernel are active FIE injects faults in CPU registers.

4 Experimental Results

This section provides the experimental results to show the SE effects in real-time applications. The SE effects on real-time kernel's services are classified as follows:

- **Safe or Masked faults**: no visible effect on system functionality.
- **Application failure**: represents a class of faults with some effects on the application level, e.g. incorrect output results, real time problems and process hanging.
- **Application Exception**: one or more applications trigger some exception routines, e.g. illegal instruction and division by zero.
- **System crash** — the system halts.

DOI: 10.1587/elex.4.755
Received September 14, 2007
Accepted October 29, 2007
Published December 10, 2007

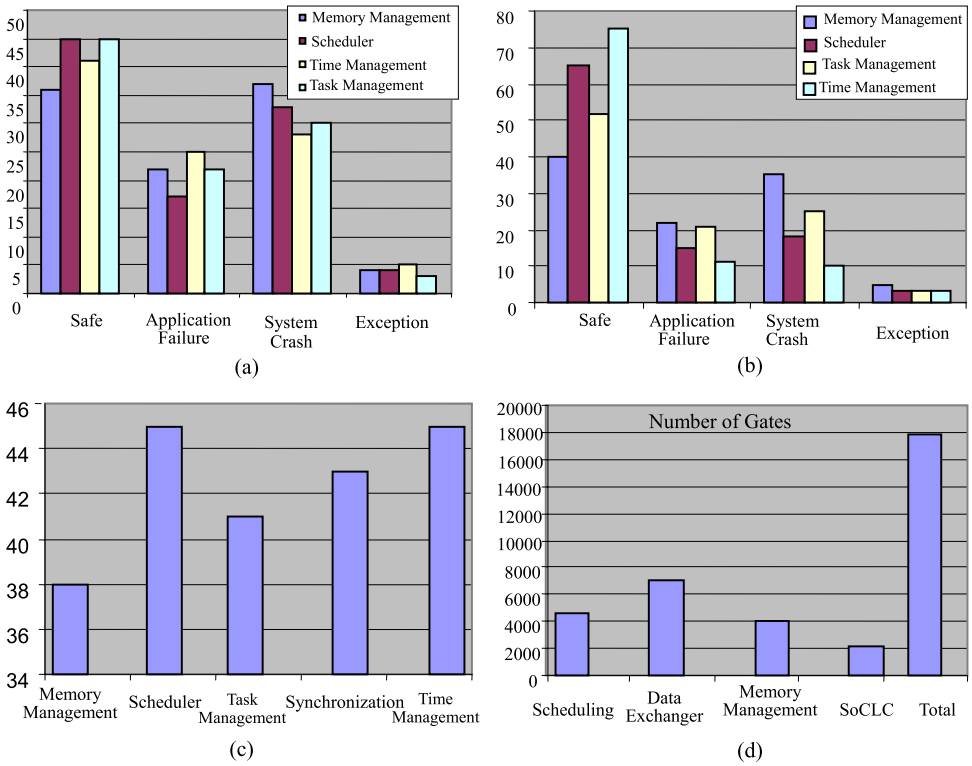


Fig. 3. (a) SE Effects in SW-RTOS; (b) SE Effects in HW/SW-RTOS; (c) Robustness of HW/SW-RTOS versus SW-RTOS in terms of SEs; (d) Hardware overhead of different HW/SW-RTOS units.

To evaluate SW-RTOS and our proposed HW/SW-RTOS assessing the reliability and different vulnerability factors for each of OS services, we performed several fault injection rules:

- During execution of SW-RTOS and our proposed HW/SW-RTOS services, faults were randomly generated by Fault Injection module and injected into the CPU registers.
- Fault Injection module will be activated by a signal coming from HW/SW-RTOS using a mechanism of data-exchanging while services are in progress.

The SE effect according to the different services which are provided by SW-RTOS and HW/SW-RTOS are illustrated in Figure 3 (a) and 3 (b). The (X) axes in these figures illustrate the classes of fault consequences, while the value axis (Y) shows the corresponding occurrences rate. Different services related to SW-RTOS and HW/SW-RTOS are depicted by column bars. For example, consequences of faults that affect services belonging to the synchronization group are illustrated by second bar from right. On average 42.4% of faults have no visible effects on the system behavior in SW-RTOS while sanguinely this factor is 57.8% in HW/SW-RTOS. The application failure rate in SW-RTOS composes 21.2% of total failure rate, but in HW/SW-RTOS this fraction improves to 16.6%. Regarding to system crashes we can see a

DOI: 10.1587/elex.4.755
Received September 14, 2007
Accepted October 29, 2007
Published December 10, 2007

15% improvement in robustness due to SEs. Having shown in Figure 3 (c), all services provided by HW/SW-RTOS are more robust than the same services provide by SW-RTOS.

Figure 3 (a) and 3 (b) shows the effectiveness of HW/SW-RTOS services in terms of reliability related to SE and the hardware overhead related to different units of HW/SW-RTOS respectively. By resorting to Figure 3 (c), we can see that both synchronization and time management services are considerably improved. These improvements can be justified by dedicated hardware synchronization part of our HW/SW-RTOS. The HW/SW-RTOS implementation has about 18000 gates of hardware overhead as shown in Figure 3 (d).

5 Conclusion

Real-time operating systems are subject to faults that affect both the correctness of logical results and the tasks' response times. Hardware-Software Real-Time Operating Systems seems to provide predictable response time at an affordable cost. In this paper, we analyzed the effect of soft-errors in real-time applications running under an RTOS which is implemented in HW/SW. The experimental results show that SEs occurring in a real-time operating system (either in SW or HW kernel) has a major impact on the system's behavior. The experimental results also show the robustness of HW/SW-RTOS services in terms of SEs against SW-RTOS services. Due to dedicated synchronization hardware, as experiments showed, we confidently obtained considerable improvements in the robustness of synchronization services provided by HW/SW-RTOS versus SW-RTOS.

DOI: 10.1587/elex.4.755
Received September 14, 2007
Accepted October 29, 2007
Published December 10, 2007