IJCSMS International Journal of Computer Science and Management Studies, Vol. 12 Issue 01, January 2012
ISSN (Online): 2231-5268
www.ijcsms.com

IMPACT OF SOFT-ERROR IN ROBUST CONFIGURABLE SYSTEM DESIGN USING REAL TIME OPERATING SYSTEM

M.Shankar[1], Dr.M.Sridar[2], Dr.M.Rajani[3]

[1]Associate Professor & Head for UG & PG studies
Department of Electrical and Electronics Engineering,
Kuppam Engineering College, Kuppam, Andhra Pradesh, India
Magaprajin@gmail.com

[2]Director International Relations
Bharath University,
Chennai, 600073, Tamilnadu, India,
deanrdinter@bharathuniv.ac.in

[3]Director of R & D,
Bharath University,
Chennai, 600073, Tamilnadu, India,
deanrd@bharathuniv.ac.in

Abstract

This paper investigates the sensitivity of real-time systems running applications under operating systems that are subject to soft-errors. A procedure for characterizing the soft error susceptibility of nodes in a logic circuit, and a heuristic procedure for selecting the set of nodes for partial duplication are described. Third, the correctness of embedded systems is currently jeopardized by soft errors that may render control systems inoperable. In general, soft errors are increasingly a problem due to smaller fabrication sizes and deployment in harsh environments. A full set of experimental results demonstrate the cost-effective tradeoffs that can be achieved. First, architects must understand the impact of soft errors on their designs. Second, they must select judiciously from among available techniques to reduce this impact in order to meet their reliability targets with minimum overhead Two reduction heuristics, cluster sharing reduction and dominant value reduction, are used to reduce the soft error failure rate significantly with a fraction of the overhead required for conventional TMR. However, FPGA's vulnerability to hard and soft errors is a major weakness to robust configurable system design. To eliminate the soft memory errors that are induced by cosmic rays, memory manufacturers must either produce designs that can resist cosmic ray effects or else invent mechanisms to detect and correct the errors.

Keywords: *Real-Time OS, soft-error, Embedded, configurable operating system.*

1. INTRODUCTION

A real-time operating system (RTOS) is an operating system that supports and guarantees timely responses to external and internal events of real-time systems An RTOS monitors, responds to, and controls an external environment, which is connected to the computer system through sensors, actuators, or other input-output (I/O) devices. In a real-time system in general and an RTOS in particular, the correctness of system behaviours depends not only on the logical results of computation but also on the time point at which the results are obtained. Real-time systems can be divided into hard and soft real-time systems. In the former, a failure to meet timing constraints will be of serious consequences, while in the latter [1], a timing failure may not significantly affect the functioning of the system. A real-time system is one whose correctness involves both the logical correctness of outputs and their timeliness. It must satisfy response-time constraints or risk severe consequences including failure. Real-time systems are classified as hard, firm or soft systems. In hard real-time systems, failure to meet response-time constraints leads to system failure.

Firm real-time systems have hard deadlines, but where a certain low probability of missing a deadline can be tolerated. Systems in which performance is degraded but not destroyed by failure to meet response time constraints are called soft real-time systems. The intelligent time slice for round robin architecture for real time operating systems is a modified version of simple round robin architecture. Simple round robin architecture cannot be implemented in real time operating systems because of high context switch rate, larger waiting time and larger response time. Because of these performance criteria of the round robin architecture is not suitable to implement in real time systems. Real time operating systems have no hard deadlines for tasks but missing of deadlines in

real time operating systems will degrade the system performance. The proposed algorithm covers all the drawbacks of round robin architecture by reducing the number of context switches, reducing the waiting time, reducing the response time thereby increasing the system throughput. The clock-driven schedulers are those in which the scheduling points are determined by the interrupts received from a clock. In the event-driven ones, the scheduling points are defined by certain events which precludes clock interrupts. The hybrid ones use both clock interrupts as well as event occurrences to define their scheduling points.

Event-driven schedulers:
• Simple priority-based
• Rate Monotonic Analysis (RMA)
• Earliest Deadline First (EDF)

We think that the use of a thread dedicated to the task scheduling makes the modelling of some scheduling policies easier, like modelling the Time Sharing algorithm for example. However, this approach increases the simulation duration since there is a context switch for each call to the scheduler and each return, what is not the case when we use procedure calls. In the case of using procedure [2] calls, the only thread switches are those of the tasks of the system we're designing that occur. Generally speaking, a real-time system is composed of a set of tasks, each running a sequential algorithm, and communicating between them with high-level communication mechanisms.

Synchronization: Based on events or Semaphore.

Message passing: Based on message queues.

Data sharing: Based on global data protected by mutual exclusion.

After having selected the basic behaviours shared by traditional RTOS, we have implemented them in a nucleus exporting a few service classes. These generic services will then serve as a founding layer for developing each emulated RTOS API, according to their own flavour and semantics. In order for this layer to be architecture neutral, the needed support for hardware control and real-time capabilities will be obtained from underlying host software architecture, through a rather simple standardized interface [3]. Thus, porting the nucleus to a new real-time architecture will solely consist in implementing this low-level interface for the target platform.

Fig.1 Basic Services Provided by a Real-Time Operating System Kernel

1.1 Classification of RTOS

RTOS's are broadly classified into three types, namely, Hard Real Time RTOS, Firm Real Time RTOS and Soft Real Time RTOS as described below:

Hard real-time: Degree of tolerance for missed deadlines is extremely small or zero. A missed deadline has catastrophic results for the system Firm real-time: missing a deadline might result in an unacceptable quality reduction

Soft real-time: Dead lines may be missed and can be recovered from. Reduction in system quality is acceptable

1.2 Features of RTOS

The design of an RTOS is essentially a balance between providing a reasonably rich feature set for application development and deployment and, not sacrificing predictability and timeliness. A basic RTOS will be equipped with the following features:

Multitasking and Preemptibility An RTOS must be multi-tasked and perceptible to support multiple tasks in real-time applications. The scheduler should be able to pre-empt any task in the system and allocate the resource to the task that needs it most even at peak load.

Task Priority Pre-emption defines the capability to identify the task that needs a resource the most and allocates it the control to obtain the resource. In RTOS, such capability is achieved by assigning individual task with the appropriate priority level. Thus, it is important for RTOS to be equipped with this feature.

Reliable and Sufficient Inter Task Communication Mechanism
For multiple tasks to communicate in a timely manner and to ensure data integrity among each other, reliable and sufficient inter-task communication and synchronization mechanisms are required [4].

Priority Inheritance

To allow applications with stringent priority requirements to be implemented, RTOS must have

a sufficient number of priority levels when using priority scheduling.

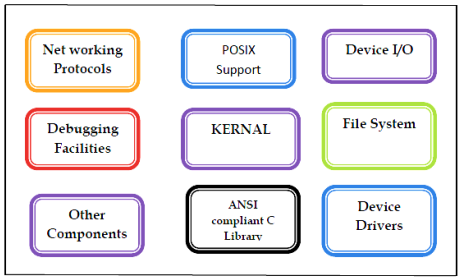

Fig.2. General Architecture of RTOS

An operating system generally consists of two parts: kernel space (kernel mode) and user space (user mode). Kernel is the smallest and central component of an operating system. Its services include managing memory and devices and also to provide an interface for software applications to use the resources. Additional services such as managing protection of programs and multitasking may be included depending on architecture of operating system. There are three broad categories of kernel models available

1.3 Monolithic kernel

A monolithic kernel is a kernel architecture where the entire kernel is run in kernel space in supervisor mode. In common with other architectures (microkernel, hybrid kernels), the kernel defines a high-level virtual interface over computer hardware, with a set of primitives or system calls to implement operating system services such as process management, concurrency, and memory management in one or more modules.

1.4 Microkernel

QNX Neutrino is based on real client/server architecture and consists of microkernel and optional cooperating processes. The microkernel implements only the core services, like threads, signals, message passing, synchronization, scheduling and timer services. Additional functionality is implemented in cooperative processes, which act as server processes and respond to the request of client processes [5]. In this case the application or any other functional modules act as a client. This technique is based on message oriented communication model. Communication model uses message bridges for message transfer to any node in the network. In that way the distributed character of QNX is deployed in its design.

1.5 Prototype Exokernels

We have implemented a prototype Exokernels system based on secure bindings, visible revocation, and abort protocols. It includes an Exokernels (Aegis) and a UN trusted library operating system (ExOS). We use this system to demonstrate several important properties of the Exokernels architecture: (1) Exokernels can be made efficient due to the limited number of simple primitives they must provide; (2) low-level secure multiplexing of hardware resources can be provided with low overhead; (3) traditional abstractions, such as VM and IPC, can be implemented efficiently at application level, where they can be easily extended, specialized, or replaced; and (4) applications can create special-purpose implementations of abstractions, tailored to their functionality and performance needs. The concept is orthogonal to that of micro- vs. monolithic kernels by giving an application efficient control over hardware. It runs only services protecting the resources (i.e. tracking the ownership, guarding the usage, revoking access to resources, etc) by providing low-level interface for library operating systems (libOSes) and leaving the management to the application.

1.6 RTOS kernel Service

Multiprocessor real-time operating system (RTOS) kernel is designed as a software platform for System on Chip (SoC) applications and hardware/software co design research purposes. This multiprocessor RTOS kernel has the key features of an RTOS, such as multitasking capabilities, event-driven priority-based pre-emptive scheduling; and interprocess communication and synchronization. The "kernel" of a real-time operating system ("RTOS") provides an "abstraction layer" that hides from application software the hardware details of the processor (or set of processors) upon which the application software will run. In doing so, it supplies five main categories of basic services to application software. The most basic category of kernel services is task management. This set of services allows application software developers to design their [6,7] software as a number of separate "chunks" of software – each handling a distinct topic, a distinct goal, and perhaps its own real-time deadline. Each separates "chunk" of software is called a "task". Services in this category include the ability to launch tasks and assign priorities to them. The main RTOS service in this category is the scheduling of tasks as the embedded system is in operation. The task scheduler controls the execution of application software tasks, and can make them run in a very timely and responsive fashion.

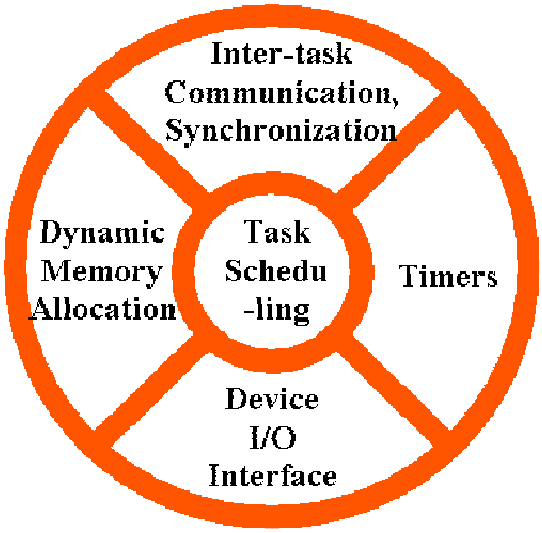

Fig. 3 RTOS Kernel Services

Embedded Configurable Operating System (eCos): eCos is provided as an open source runtime system supported by the GNU open source development tools. Developers have full and unfettered access to all aspects of the runtime system. No parts of it are proprietary or hidden, and you are at liberty to examine, add to, and modify the code as you deem necessary. These rights are granted to you and protected by the eCos license. It also grants you the right to freely develop and distribute applications based on eCos. We welcome all contributions back to eCos such as board ports, device drivers and other components, as this helps the growth and development of eCos, and is of benefit to the entire eCos community. One of the key technological innovations in eCos is the configuration system.

The configuration system allows the application writer to impose their requirements on the run-time components, both in terms of their functionality and implementation, whereas traditionally the operating [8,9] system has constrained the application's own implementation. Essentially, this enables eCos developers to create their own application-specific operating system and makes eCos suitable for a wide range of embedded uses. Configuration also ensures that the resource footprint of eCos is minimized as all unnecessary functionality and features are removed. The configuration system also presents eCos as component architecture. This provides a standardized mechanism for component suppliers to extend the functionality of eCos and allows applications to be built from a wide set of optional configurable run-time components. Components can be provided from a variety of sources including the standard eCos release, commercial third party developers and open source contributors.

2. Literature survey

System-on-a-chip (SoC) technology is the packaging of all the necessary electronic circuits and parts for a "system" (such as a cell phone or digital camera) on a single integrated circuit (IC), generally known as a microchip . For example, a system-on-a-chip for a sound-detecting device might include an audio receiver, an analog-to-digital converter (ADC), a microprocessor , necessary memory , and the input/output logic control for a user - all on a single microchip. System-on-a-chip technology is used in small, increasingly complex consumer electronic devices.

Some such devices have more processing power and memory than a typical 10-year-old desktop computer. In the future, SoC-equipped nano robots of microscopic dimensions) might act as programmable antibodies to fend off previously incurable diseases. SoC video devices might be embedded in the brains of blind people, allowing them to see; SoC audio devices might allow deaf people to hear. Handheld computers with small whip antennas might someday be capable of browsing the Internet at megabit-per-second speeds from any point on the surface of the earth.

We were motivated to measure the area, performance and power consumption gap between field-programmable gate arrays (FPGAs) and standard cell application-specific integrated circuits (ASICs) for the following reasons: In the early stages of system design, when system architects choose their implementation medium, they of- ten choose between FPGAs and ASICs. Such decisions are based on the differences in cost (which is related to area), performance and power consumption between these implementation media but to date there have been few attempts to quantify these differences. A system architect can use these measurements to access whether implementation in an FPGA is feasible.

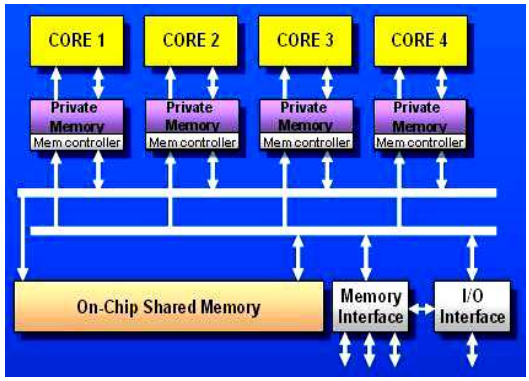

Fig 4.Multiprocessor System-on-a-Chip [MPSoC]

MPSoC architecture has become an unavoidable part of designing embedded systems dedicated to applications that requires intensive parallel computations. The most important design challenge in such systems consists in solving the huge architectural space solution and evaluating the corresponding alternatives. MPSoC systems need new development methodology to reduce the complexity of design space exploration and to increase the engineer's productivity. A strong advantage of using the MARTE profile for MPSoC modelling is the particular concept of factorization, both for hardware architecture and application. With the semantic introduced by the ARRAYOL model of computation [Bou07], factorization provides a mechanism that expresses the parallelism of the system in a compact way.

3. Hardware/Software RTOS Design

3.1. RTOS/MPSOC

Our MPSoC is most similar to Raw, as the processors themselves are simple 5-stage RISCs, without support for SIMD operations and wide register files; however, our use of AVC buffers is similar to the SRF/LRF and scratchpad memories, but for processor-to-processor communication. For the remainder of the paper we use the JPEG compression algorithm as motivational example and case study. Although JPEG compression is a relatively simple algorithm, it is easy to understand, and representative for streaming applications [10,11]. This paper advocates the instantiation of application-specific double buffers between adjacent cores in MPSoC in order to enhance memory system performance for stream programs. We refer to each double buffer as a single *Architecturally Visible Communication (AVC)* buffer. AVC buffers can be viewed as a scratchpad memory that is shared between two cores

3.2. Hardware RTOS components

Our solution is provided in the form of an intellectual property (IP) hardware unit which we call the SoC Lock Cache (SoCLC). The SoCLC provides elective lock hand-o® by reducing on-chip memory trace and improving performance in terms of lock latency, lock delay and bandwidth consumption. In our methodology, lock variables are accessed via SoCLC hardware. The SoCLC consists of one-bit registers to store lock variables and associated control logic to electively implement the lock hand-o® via interrupt generation, which eliminates busy-wait problems. In this way, the SoCLC eliminates the use of the main memory bus for unnecessary spinning and thus enables the memory bandwidth to be available for other useful work [12,13]. The SoCLC has been shown to achieve speedups of 55% and 27% in realistic examples when compared to the traditional spin-lock mechanism at a very small (<13,000 gates) hardware cost [1], [2], [3]. However, it is also desired to be able to customize/ configure and parameterize (according to the customer specifications) the SoCLC with the minimum engineering effort possible in an automated fashion. One approach to solve these demands can be referred to as an IP-generator tool. In this context, we present PARLAK, parameterized lock cache generator, that generates a custom SoCLC for an SoC including reconfigurable and/or custom logic and multiple heterogeneous processors.

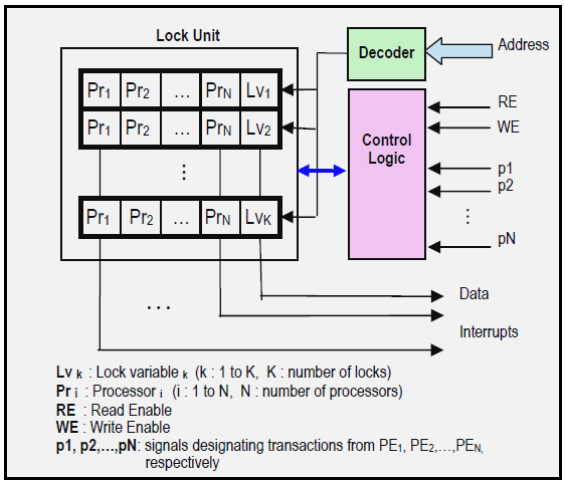

Fig. 5 Soc Architecture

3.3 SoCDMMU

In static memory management, the memory is allocated (or assigned) at compile (or design) time. Static memory management can be as simple as allocating static arrays or as complex as synthesizing memory structures and/or software memory allocators suitable for certain type of applications. In dynamic memory management, memory is allocated at run-time. Dynamic memory management uses memory efficiently when compared to static memory management techniques. However, dynamic memory management can consume a great amount of a program's execution time [41] {especially in object-oriented applications [14,15]. Dynamic memory management can be classified into two categories: _ Manual memory management In manual memory management, the programme has direct control over when memory is allocated and when it might be recycled. Usually this is either by explicit calls to heap management functions (e.g., malloc ()/free () in the C language) or by language constructs that stack (such as local variables). Although manual memory management is easier for programmers to understand and use, memory

management bugs are common when manual memory management is used. _ Automatic memory management Automatic memory management is a service, either as a part of the language (e.g., Java and Lisp) or as an extension, that automatically deallocates memory that a program will not use again. Automatic memory managers (often known as garbage collectors) usually do their job by recycling blocks that are unreachable from program variables

4. Experimental framework

RTOS's are broadly classified in to three types, namely, the Hard Real Time RTOS, Firm Real Time RTOS and Soft Real Time RTOS, which are describes bellow.

Hard real-time: missing a deadline has catastrophic results for the system;

Firm real-time: missing a deadline entails an unacceptable quality reduction as a consequence

Soft real-time: deadlines may be missed and can be recovered from. The reduction in system quality is acceptable

4.1. Hard real time Operating System

Hard real-time is that a small high-priority real-time kernel runs between the hardware and standard Linux. The real-time tasks are executed by this real-time kernel (run to completion) and normal Linux processes are suspended during this duration. The real-time scheduler of the real-time kernel treats the standard Linux kernel as an idle task, which when given a chance to run, executes its own scheduler to schedule normal Linux processes. But since the real-time kernel runs at a higher priority, the normal Linux processes can at any time be pre-empted by a real-time task.

4.2 Software Real Time Operating System (SW-RTOS)

Soft real time means that only the precedence and sequence for the task operations are defined, interrupt latencies and context switching latencies are small but there can be few deviations between expected latencies of the tasks and observed time constraints and a few deadline misses are accepted. The pre-emption period for the soft real time task in worst case may be about a few ms. Mobile phone, digital cameras and orchestra playing robots are examples of soft real time systems.

5. Experimental results

This section describes and analyses the obtained results to get evidence of soft-errors consequences in the case of a real-time application. Transient faults may cause several malfunctions when the real-time kernel's services are corrupted. These malfunctions are classified as follows:

Effect less – no visible effect on system functionality Exception trigger – the program triggers some exception routine (e.g. illegal instruction, division by zero, etc.);

- System crash – the system stops functioning;
- Application failure – represents a class of faults with visible consequences on the application level. This class of faults can be subdivided as:
- Incorrect output results – one or more application tasks are able to provide results, but they are different from the expected ones;
- Real-time problem – one or more application tasks do not respect their real-time constraints;
- Task hang – system still works but one or more application tasks stop functioning

5.1 Fault parameters generator

Calculates when and where the fault will be injected. FIM can inject faults in CPU registers, cache memories or internal and external memories, but in our experiments, faults consist of single bit-flips only in the CPU registers while the main services of the Micro C kernel are active.

5.2 Fault tracer

Collect the information about the currently executed task. At each task execution, it saves into a file the instants when the task starts and ends its execution as well as the tasks' output results,

5.3 Results analyzer

Uses information provided by the fault tracer module in order to classify the fault consequences µC/OS-II is a portable, ROM able, scalable, preemptive, real-time deterministic multitasking kernel for microprocessors, microcontrollers and DSPs. Offering unprecedented ease-of-use, µC/OS-II is delivered with complete 100% ANSI C source code and in-depth documentation. µC/OS-II runs on the largest number of processor architectures, with ports available for download from the Marcum Web site. µC/OS-II manages up to 250 application tasks. µC/OS-II includes: semaphores; event flags; mutual-exclusion semaphores that eliminate unbounded priority inversions; message mailboxes and queues; task, time and timer

management; and fixed sized memory block management. µC/OS-II's footprint can be scaled (between 5 Kbytes to 24 Kbytes) to only contain the features required for a specific application. The execution time for most services provided by µC/OS-II is both constant and deterministic; execution times do not depend on the number of tasks running in the application. To assess the MicroC kernel sensitivity to transient faults, we performed several fault injection campaigns.

Faults were randomly injected in the CPU registers during execution of MicroC services. Note that, we did not consider the issue of injecting faults in the MicroC services that are used only for system initialization (e.g. the *task creation service* that is used only when the application tasks are created). The impact of soft-errors according to the different groups of MicroC services is illustrated. The category axis (X) illustrates the classes of fault consequences, while the value axis (Y) shows their respective occurrence frequency. The different groups of MicroC services are depicted by a column bar [9]. For instance, consequences of faults that affect services belonging to the time management group are illustrated by the column bar with vertical lines pattern (the 4th column in each class of fault consequences).

This section describes and analyses the obtained results to get evidence of soft-errors consequences in the case of a real-time application. Transient faults may cause several malfunctions when the real-time kernel's services are corrupted.

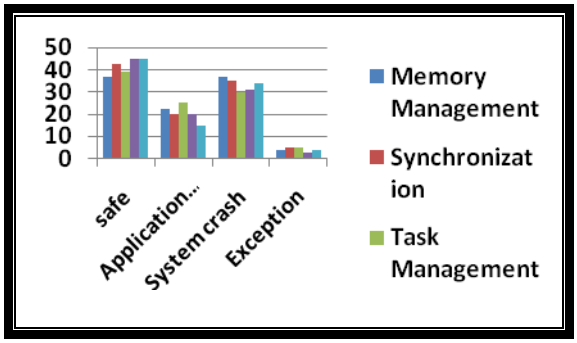

Fig. 6 Effect of Soft-Error in SW-RTOS

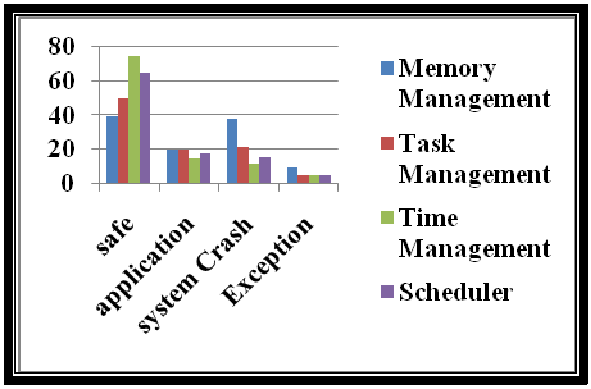

Fig. 7 Effect of Soft-Error in HW/SW-RTOS

• Process Hanging (system continue its working but some processes stop their operations).

• Application Exception: one or more application tasks trigger some exception routine (e.g. illegal instruction, division by zero and etc.).

• System crash - the system stops functioning. Fault Injection Results to evaluate eCos (SW-RTOS) and their proposed HW/SW-RTOS assessing the reliability and different vulnerability factor (VF) for each of OS services, we performed following fault injection rules:

I. During execution of (SW-RTOS) and their proposed HW/SW-RTOS services, faults were randomly generated by Fault Injection module then injected into the CPU registers.

II. Operating System Services like task creation and [16] task termination are safe to fault injection. During these services Fault Injection module is idle. Fault Injection module will be activated using signal from HW/SW-RTOS by mechanism of data-exchanging while services are in progress.

The impact of soft-errors according to the different services which are provided by eCos (SW-RTOS) and HW/SW-RTOS based on eCos as illustrated above. The X axes in these Fig it illustrate the classes of fault consequences that were specified before, while the value axis (Y) shows their frequency of occurrence. The different groups services related to eCos (SW-RTOS) and HW/SWRTOS are depicted by a column bar. For instance, consequences of faults that affect services belonging to the synchronization group are illustrated by orange color. On average 42.4% of faults have no visible effects on the system behaviour in SW-RTOS in comparison with 57.8% of fault have no effect in HW/SW-RTOS. Application failure rate SW-RTOS consist of 21.2% of total failure rate but in HW/SW-RTOS this fraction [17] improves to 16.6%. Regarding to

system crashes we can see a 15% improvement in robustness due to soft-error. A remarkable feature of their results that is apparent from that all services provided by HW/SWRTOS are more robust than the same services provided by Services related to both synchronization and time managements are considerably improved as shown in Fig. These improvements can be justified by dedicated hardware synchronization part of our HW/SW-RTOS.

6. Conclusion

Nowadays, safety-critical applications are often based on real-time operating systems. These systems are subject to faults that affect both the correctness of logical results and the timing of tasks response. In this paper, we reported a detailed analysis of soft-errors impact on the key services of MicroC, taking into account the application timing constraints. Our results show that soft-errors occurring in a real time operating system's kernel have a major impact on the system's behaviour. Moreover, it was found that all groups of MicroC services have the same sensitivity profile. Solutions to the soft error problem will be the increasing set of embedded applications in harsh environments, which comprise critical infrastructure in today's society. This is particularly true for to aircraft using commodity microprocessors for control systems.

6. References

[1] Douglass, B. P. Doing hard time: Developing real-time systems with UML, objects, frameworks, and patterns. Addison-Wesley, 1999.

[2] D. Mossé, R. Melhelm, S. Ghosh, "A non-pre-emptive real-time scheduler with recovery from transient faults and its implementation" IEEE Transactions on Software Engineering, Vol. 29, No. 8, August 2003, pp 752-767

[3] J.-C. Fabre, F. Salles, M. Rodriguez, J. Arlat, "Assessment of COTS Microkernel's by fault injection", Dependable Computing for Critical Applications 7, San- Jose, California, USA, 6-8 January 1999, pp.25-44

[4] J. Arlat, J.-C. Fabre, M. Rodriguez, "Dependability of COTS microkernel-based systems", IEEE Transactions on Computers Volume 51, No 2, Feb. 2002, pp. 138 – 163 [5] W.-L Kao, D Tang, R.K. Iyer, "Study of fault propagation using fault injection in the UNIX system", 2nd Test Symp. 16-18 Nov., 1993, pp. 38-43

[5] V.Narayanan and Yuan Xie, .Reliability concerns in embedded system designs,. IEEE Computer magazine, pp. 106.108, January, 2006.

[6] Hisashige Ando, Yuuji Yoshida, Aiichiro Inoue, Itsumi Sugiyama, Takeo Asakawa, Kuniki Morita, in Design Automation Conference, New York, NY, USA, 2003, pp. 702.705, ACM Press.

[7] Y.C. Yeh, .Triple-triple redundant 777 primary light computer,. in 1996IEEE Aerospace Applications Conference. Proceedings, 1996, vol. 1, pp. 293.307.

[8] Y. C. (Bob) Yeh, .Design considerations in boeing 777 wwire-by-wire computers, .in IEEE International High-Assurance Systems Engineering Symposium, 1998, p. 64.

[9] Joel R. Sklaroff, .Redundancy management technique for space shuttle computers,. IBM Journal of Research and Development, vol. 20, no. 1, pp. 20.28, 1976

[10] P. L. Murray, "Re-Programmable FPGAs in Space Environments".

[11] IEEE Standard Test Access Port and Boundary Scan Architecture (IEEE Std 1149.1), IEEE Std. Board, 2001. [12] J. H. Lala; R. E. Harper, "Architectural principles for safety-critical real-time applications," Proceedings of the IEEE, Vol. 82, No. 1, pp. 25-40, January 1994

[12] J.W.S. Liu,W.-K. Shih, K.-J. Lin, R. Bettati, J.-Y. Chung, "Imprecise Computations", Proceedings of the IEEE, Vol.82, No.1, Jan. 1994, pp.83-93

[13] P. Mejia-Alvarez, D. Mossé, "A responsiveness approach for scheduling fault-recovery in real-time systems", 5th Real-Time Technology and Applications Symposium, 2-4 June 1999, pp.4-13

[14] M. Rodriguez, A. Albinet, J. Arlat, "MAFALDA-RT: a tool for dependability assessment of real-time systems", International Conference on Dependable Systems and Networks, USA, 23-26 June 2002, pp. 267 – 272

[15] B. Nicolescu, N. Ignat, Y. Savaria, 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, Sept. 19-23, 2005

[16] Ph. Shirvani, R. Saxena, E.J. McCluskey, "Software implemented EDAC protection against SEUs", IEEE Transaction on Reliability, Vol. 49, No. 3, Sept. 2000, pp. 273-284

[17] V. Izosimov, P. Pop, P. Eles, Z. Peng, "Design optimization of time- and cost-constrained fault-tolerant distributed embedded systems", Design, Automation and Test in Europe, Munich, Germany, 7-11 Mars 2005, pp. 864-869