An Approach to Advance Real-Time Os Services with Soft-Error

Swapan Debbarma

***Abstract*—the rises of RTOSs have evolved from single-use specialized systems to a wide of more general purpose Operating Systems (such as real-time variants of Open Source Operation System. These-days, numbers of critical applications that have stringent real-time constraint are placed and run in an environment with Real-Time operating system (RTOS). The provided services of RTOSs are subject to faults that affect both functional and timing of Tasks which are running based on RTOS. In this paper, we try to evaluate and analyze robustness of services due to soft-errors in two proposed architecture of RTOS which are (SW-RTOS and HW/SW-RTOS). Accordingly, we propose for an architecture which provides more robust services in term of soft-error. Real-Time Operating System (RTOS) users desire predictable response time at an affordable cost, due to this demand Hardware/Software Real-Time Operating Systems (HW/SW-RTOS) appeared. This paper analyzes the impact of soft-errors in real-time systems running applications under purely Software RTOS versus HW/SW-RTOS. The proposed model is used to evaluate robustness of services like scheduling, synchronization time management and memory management and inter process communication in Software based RTOS and HW/SW-RTOS. Experimental results show HW/SW-RTOS provide more robust services in term of soft-error against purely software based RTOS.**

***Index Terms*—Real-Time OS; soft-error; embedded conFig.urable operating system (eCOS)**

I. Introduction

A real-time operating system (RTOS) is a multitasking operating system intended for real-time applications. Such applications include embedded systems (programmable thermostats, household appliance controllers), industrial robots, spacecraft, industrial control, and scientific research equipment.

An RTOS does not necessarily have high throughput; rather, an RTOS provides facilities which, if used properly, guarantee deadlines can be met generally or deterministically (known as soft or hard real-time, respectively). An RTOS will typically use specialized scheduling algorithms in order to provide the real-time developer with the tools necessary to produce deterministic behavior in the final system. An RTOS is valued more for how quickly and/or predictably it can respond to a particular event than for the amount of work it can perform over a given period of time.

Two basic designs exist:

- Event-driven (priority scheduling) designs switch tasks only when an event of higher priority needs service, called pre-emptive priority.
- Time-sharing designs switch tasks on a clock interrupt, and on events, called round robin.

Manuscript received February24, 2011; revised July 6, 2011.

Time-sharing designs switch tasks more often than is strictly needed, but give smoother, more deterministic multitasking, giving the illusion that a process or user has sole use of a machine.

Early CPU designs needed many cycles to switch tasks, during which the CPU could do nothing useful, so early operating systems tried to minimize wasting CPU time by maximally avoiding unnecessary task-switches.

Real-Time Operating System ("RTOS") provides an "abstraction layer" that hides hardware details of processor (or set of processors) from software layer .In providing this "abstraction layer" the RTOS kernel supplies four main types of basic services to application software.

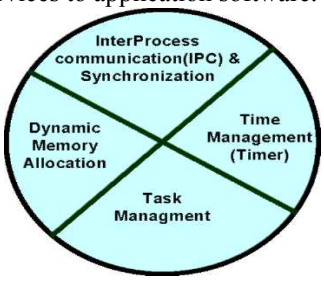

Fig.1. Basic Services Provided by a Real-Time Operating System Kernel

A. RTOS architecture

The architecture of an RTOS is dependent on the complexity of its deployment. Good RTOSs are scalable to meet different sets of requirements for different applications. For simple applications, an RTOS usually comprises only a kernel. For more complex embedded systems, an RTOS can be a combination of various modules, including the kernel, networking protocol stacks.

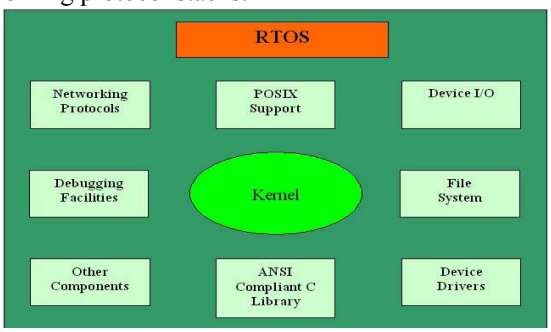

Fig.2. General Architecture of RTOS

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,

namely:

- Monolithic kernel:

It runs all basic system services (i.e. process and memory management, interrupt handling and I/O communication, file system, etc) in kernel space. As such, monolithic kernels provide rich and powerful abstractions of the underlying hardware. Amount of context switches and messaging involved are greatly reduced which makes it run faster than microkernel. Examples are Linux and Windows.

- Microkernel:

It runs only basic process communication (messaging) and I/O control. The other system services (file system. networking, etc) reside in user space in the form of daemons/servers. Thus, micro kernels provide a smaller set of simple hardware abstractions. It is more stable than monolithic as the kernel is unaffected even if the servers failed (i.e. File System). Examples are AmigaOS and QNX.

- Exokernel:

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.

A. RTOS kernel Service:

An RTOS generally avoids implementing the kernel as a large monolithic program. The kernel is developed instead as a micro-kernel with added conFig.urable functionalities. This implementation gives resulting benefit in increase system conFig.urability, as each embedded application requires a specific set of system services with respect to its characteristics. The kernel of an RTOS provides an abstraction layer between the application software and hardware. This abstraction layer comprises of six main types of common services provided by the kernel to the application software.

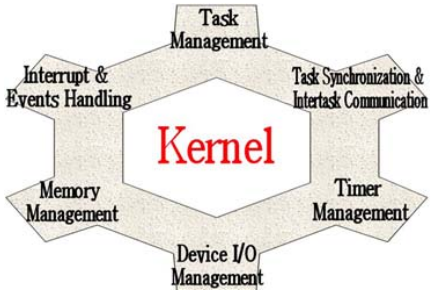

Fig. 3. RTOS Kernel Services

B. Embedded ConFig.urable Operating System (eCos):

Egos (embedded conFig.urable operating system) is an open source, royalty-free, real-time operating system intended for embedded systems and applications which need only one process with multiple threads. eCos was designed for devices with memory size in the tens to hundreds of kilobytes, or with real-time requirements. eCos runs on a wide variety of hardware platforms, including ARM, Caloric, FR-V, Hitachi H8, IA-32, Motorola 68000, Matsushita AM3x,MIPS, NEC V8xx, Nikos II, PowerPC, SPARC, and Super.

II. Litarature Survey

Traditionally, an Operating System (OS) implements in software basic system functions such as task/process management and I/O. Furthermore, a Real-Time Operating Systems (RTOS) has also been implemented in software to manage tasks in a predictable, real-time manner. However, with System-on-a-Chip (SoC) architectures which is similar and more common, OS and RTOS functionality need not be implemented solely in software. Thus, partitioning the interface between hardware and software for an OS is a new idea that can have a significant impact.

Recent trends in chip design press the need for more advanced operating systems for System-on-a-Chip (SoC). However, unlike earlier trends where the focus was on scientific computing, today's SoC designs tend to be driven more by the needs of embedded computing. While it is hard to state exactly what constitutes embedded computing, it is safe to say that the needs of embedded computing form a superset of scientific computing. For example, real-time behavior is critical in many embedded platforms due to close interaction with non-humans, e.g., rapidly moving mechanical parts. In fact, the Application-Specific Integrated Circuits (ASICs) preceding SoC did not integrate multiple processors with custom hardware, but instead were almost exclusively digital logic specialized to a particular task and hence very timing predictable and exact. Therefore, we predict that advances in operating systems for SoC focusing on Real-Time Operating System (RTOS) design provide a more natural evolution for chip design as well as being compatible with real-time systems.

The technologies of Multiprocessor System-on-a-Chip (MPSoC) and reconFig.-urable chips, many hardware Intellectual Property (IP) cores that implement software algorithms have been developed to speed up computation and utilize low cost hardware. However, fully exploiting these innovative hardware IP cores have had many difficulties such as interfacing IP cores to a specific system, modifying IP cores to fulfill requirements of a system under consideration, porting device drivers and finally integrating both IP cores and software seamlessly. Much work of interfacing, modifying and/or porting IP cores and device drivers has relied on human resources. Hardware/software codesign frameworks can help reduce the burden on designers.

III. Hardware/Software Rtos Design

B. RTOS/MPSOC

Our primary target is the MPSoC, which consist of multiple processing elements with L1 caches, a large L2 memory, and multiple hardware IP components with essential interfaces such as a memory controller, an arbiter and a bus system. The target also has a shared memory multiprocessor RTOS (Atalanta developed at the Georgia Institute of Technology), which is small and conFig.urable. The code of Atalanta RTOS version 0.3 resides in shared memory, and all processing elements (PEs) execute the same RTOS code and share kernel structures as well as the states of all processes and resources. Atalanta supports priority scheduling with priority inheritance as well as

round-robin; task management such as task creation, suspension and resumption; various Inter Process Communication (IPC) primitives such as semaphores, mutexes, mailboxes, queues and events; memory management; and interrupts. The hardware IP cores can be either integrated into the reconFig.urable logic or implemented as custom logic. Besides, specialized IP cores such as DSP processors and wireless interface cores can also be integrated into the chip.

C. Hardware RTOS components:

This subsection briefly mentions two available hardware components: SoCLC and SoCDMMU.

System on chip lock cache (SoCLC)

Synchronization has always been a critical issue in multiprocessor systems. As multiprocessors execute a multitasking application on top of an RTOS, any important shared data structure, also called a Critical Section (CS), may be accessed for inter-process communication and synchronization events occurring among the tasks/processors in the system.

Previous work has shown that the System-on-a-Chip Lock Cache (SoCLC), which is a specialized custom hardware unit realizing effective lock-based synchronization for a multiprocessor shared-memory SoC, which reduces on-chip memory traffic, provides a fair and fast lock hand-off,

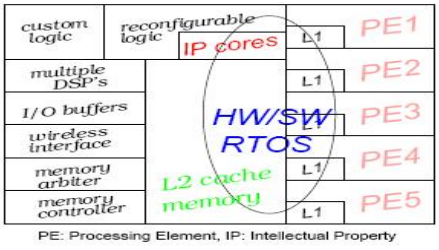

Fig. 3.1. Proposed HW/SW-RTOS

Simplifies software, increases the real-time predictability of the system and improves performance as well.

The SoCLC mechanism with a priority inheritance supports implementation in hardware. Priority inheritance provides a higher level of real-time guarantees for synchronizing application tasks. It present a solution to the priority inversion problem in the context of an MPSoC by integrating an immediate priority ceiling protocol (IPCP) implemented in hardware. The approach also provides higher performance and better predictability for real-time applications running on an MPSoC.

- SoCDMMU

The System-on-a-Chip Dynamic Memory Management Unit (SoCDMMU) is a hardware unit that allows a fast and deterministic way to dynamically allocate/deallocate global (L2) memory among PEs. The SoCDMMU is able to convert the PE address (virtual address) to a physical address. The memory mapped address or I/O port to which the SoCDMMU is mapped is used to send commands to the SoCDMMU (writing data to the port or memory-mapped location) and to receive the results of the command execution (reading from the port or memory-mapped location).

The SoCDMMU achieves a 4.4X overall speed-up in memory management during the application transition time when compared to conventional memory allocation/deallocation techniques, i.e., malloc() and free(). The SoCDMMU is synthesizable and has been integrated into a system example including porting SoCDMMU functionality to an RTOS (so that the user can access SoCDMMU functionality using standard software memory management APIs). Also, the SoCDMMU-crossbar (Xbar) switch Generator (DX-Gt) can conFig. and optimize the SoCDMMU to suit a specific system (e.g., for a particular memory conFig.uration and number of PEs). In this way, DX-Gt automates the customization and the generation of the hardware memory management functionalities.

IV. Experimental Framework

This section presents the approach for both designing a HW/SW-RTOS and injecting faults in the proposed model.

- Software Real Time Operating System (SW-RTOS):

In this Frame work one can use eCos (embedded ConFig.urable operating system) as Purely Software based RTOS (SW-RTOS). eCos is an open source, royalty-free and real-time operating system intended for embedded systems and applications. The highly conFig.urable nature of eCos allows the operating system to be customized to precise application requirements, delivering the best possible run-time performance and an optimized hardware resource footprint. A thriving net community has grown up around the operating system ensuring on-going technical innovation and wide platform support. eCos was designed for devices with memory footprints in the tens to hundreds of kilobytes, or with real-time requirements. It can be used on hardware that doesn't have enough RAM to support embedded Linux, which currently requires a minimum of about 2 MB of RAM, not including application and service requirements.

- Hardware/Software Real-Time Operating System (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 their proposed HW/SW-RTOS implementation, they 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 their proposed HW/SW-RTOS, they 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.Fig.4. shows the comparison between standard SW-RTOS (top) and their proposed

HW/SW-RTOS architecture (bottom). As in Fig. 3.1, in the proposed HW/SW-RTOS, the operating system is composed of five units:

- Scheduling unit (Implemented of HW)
- Data Exchanger unit (Implemented of HW)
- System on Chip Lock Cache unit (Implemented of HW)
- Context Switching unit (Implemented of SW)
- Memory Management unit (MMU)

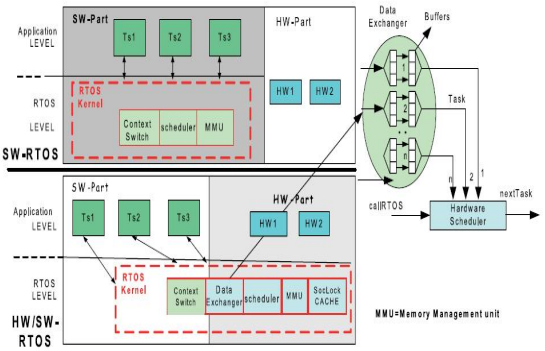

Fig. 4. SW-RTOS versus HW/SW-RTOS

- Scheduling unit

The scheduling unit finds out the next software task ID. In their 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.

- Data exchanger unit

Data exchanger unit uses buffers to pass data between different tasks. When a task tries to send data to another task, it informs Data Exchanger unit identifier of destination task and the value which must be sent. In this case Data Exchanger manages internal buffers to guarantee that the value will be reached to the specified task. Conversely, when a task needs data provided by some other tasks, it informs Data Exchanger unit identifier of source task. Then Data Exchanger blocks waited task and calls scheduler unit to send the identifier of the schedulable task to the CPU.

- System on chip lock cache unit (SoCLC)

The System on chip lock cache (SoCLC) is used in the framework to handle mutual exclusions as shown in Fig. 3.2. In this method, if process p1 executes wait (L1) instruction, it will wait until L1 is activated. To do so, p1 will be added to the list of waited processes for L1. As a result of the execution of this instruction, L' and L" will be blocked until another process executes signal (L1) instruction. However, Triple Module Redundancy (TMR) may be used to keep three dedicated lists for each lock variable to mitigate the Soft-Error effects.

- Context switching

Typical context switching consists of three steps i.e.

1) Pushing all CPU registers to the current task stack,
2) Scheduling the next task to be run,
3) Popping all CPU registers to the next task.

The steps 1 and 3 are implemented in 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.

- Fault Injection Environment (FIE)

The fault injection technique was initially proposed, but up to now cannot find any fault injection mechanism in the literature, which can support fault-injection in SW-RTOS as well as HW/SW-RTOS. Before describing main features of the adopted fault injection mechanisms, it is worth to know that faults consist of single bit-flips only in the CPU registers, therefore in order to identify sensitive components of the studied real-time operating systems, fault injection module should keep access to CPU registers to inject fault during the execution of applications.

The adopted system architecture is simulated with an Instruction Set Simulator (ISS). The fault injection tool uses temporal breakpoint features available in the ISS to inject faults by software means. Once a temporal breakpoint is reached, global execution is suspended and the ISS activates a Fault Injection Environment (FIE) that comprises two modules as shown below.

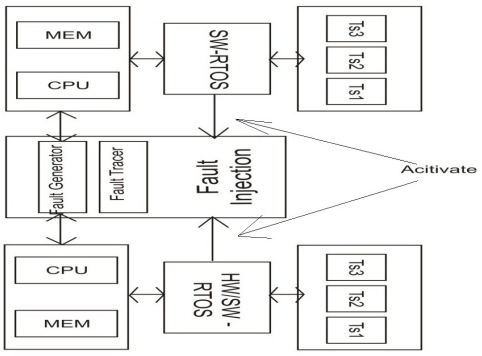

Fig. 5. Fault Injection Environment

Fault generator - calculates when and where the fault will be injected. FIE can inject faults in CPU registers, while the main services of the SW-RTOS (eCos kernel) and HW/SW-RTOS kernel are active.Fault tracer- collects information about the services that are currently executing in RTOS, (SW-RTOS) and (HW/SW-RTOS) inform this part of (FIE) about the kind of services that are active.

- Characteristics of bench mark

In order to perform the experiments, the researchers have created multitask applications classified in 6 groups according to the mechanisms through which they can communicate and synchronize. These groups fully exploit most important services offered by eCos real-time kernel. The studied application consists of six groups. Before performing any fault injection experiment, they carefully studied and tested these applications in an environment without soft-errors to verify that all tasks meet their deadlines and produce correct output results. In order to consider real-time constraint, all tasks are considered critical (we must complete their executions before their deadlines and produce logically correct responses) and perform some useful computations.

- Group1 - tasks T1, T2, T3 and T4 share global variables using semaphore, this technique can be used to access critical section and synchronization.

Fig. 6. Group 1

- Group2 - tasks T1 and T2 communicate by message queues. T1 (transmitter) sends the results of its computations into QM message queues. T2 (receiver) reads messages from T1, and uses them in their context.

Fig. 7. Group 2

- Group3 - tasks T1 and T2 communicate by a mailbox that can store a single message. T1 sends a message periodically into a mailbox, while T2, the receiver, consumes the message and uses it in its future operations.

Fig. 8. Group 3

- Group4 - tasks T1, T2 and T3 access a global variable which has been protected by mutual exclusion semaphore (mutex).

Fig. 9. Group 4

- Group5 - tasks T1, T2, T3, and T4 access a global variable using semaphore1 (sem1), while tasks T5, T6, T7, and T1 access global variable using semaphore2 (sem2).

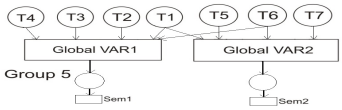

Fig. 10. Group 5

- Group6 - tasks T1, and T2 access a global variable using mutex; then each of them, that gain access to global variable, sends the results of its computations into message queue (QM) and finally, task T3 receives its message from message queue (QM).

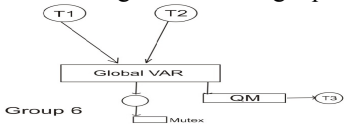

Fig. 11. Group 6

V. 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:

- Safe: no visible effect on system functionality.
- Application failure: represents a class of faults with some effects on the application level. This class of faults can be subdivided to:

i) Incorrect output results,
ii) Real-time problem
iii) 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:

- 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.
- Operating System Services like task creation and 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.s 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/SW-RTOS 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 behavior 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 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 Fig. 6. and 7. is that all services provided by HW/SW-RTOS are more robust than the same services provided by eCos (SW-RTOS).

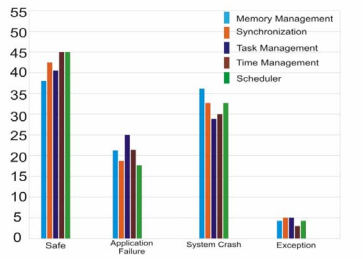

Fig. 12. Effect of Soft-Error in SW-RTOS

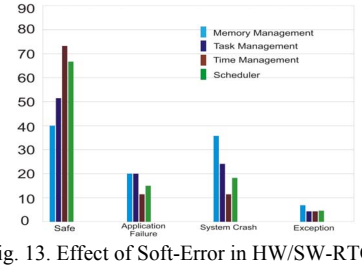

Fig. 13. Effect of Soft-Error in HW/SW-RTOS

Fig. 14. Shows the effectiveness of HW/SW-RTOS services in terms of reliability related to soft-error.

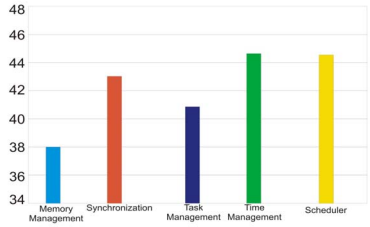

Fig. 14. Robustness of HW/SW-RTOS versus SW-RTOS .

Services related to both synchronization and time managements are considerably improved as shown in Fig. 8. These improvements can be justified by dedicated hardware synchronization part of our HW/SW-RTOS.

Fig. 15. Shows the hardware overhead related to different units of HW/SW-RTOS. As shown in this Fig., the HW/SW-RTOS implementation imposed us hardware overhead equals to 17830 gates.

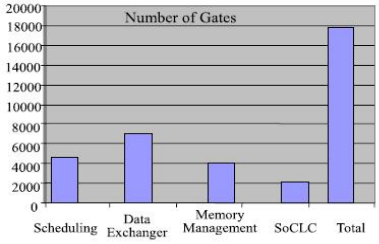

Fig. 15. Hardware overhead of different HW/SW-RTOS

VI. Conclusion

Real-time applications which have safety-critical constraints are often based on real-time operating systems. Real-time operating systems are subject to faults that affect both the correctness of logical results and the timing of tasks response.

Hardware Real-Time Operating Systems (HW/RTOS) appeared to provide predictable response time at an affordable cost. In this report, the impact of soft-error in real-time applications running under a RTOS is analyzed which is implemented in HW/SW. Their experimental results show that soft-errors occurring in a real-time operating system (either in SW or HW kernel) have a major impact on the system's behavior. Moreover, it was found that all groups of eCos services have the same sensitivity profile. Experimental results also show the robustness of HW/SW-RTOS services in term of soft-error versus SW-RTOS services. Experiments show considerable improvement in robustness of synchronization services which are provided by HW/SW-RTOS against SW-RTOS, due to the dedicated synchronization hardware.

References

[1] Yaashuwanth C and Dr. R. Ramesh"International Journal of Computer and Electrical Engineering, Vol.2, No.6, December, 2010 1793-8163

[2] V.J. Mooney and D.M. Blough, "A hardware software real-time operating system framework for socs," IEEE Des. Test vol.19, no. 6, pp 44-51, 2002.

[3] "Hardware Software partitioning of operating systems: focus on deadlock detection and avoidance," Jaehwan John Lee and Vincent John Mooney III.

[4] "Hardware Software partitioning of operating systems," Vincent J Mooney III.

[5] V.Nollet, T.Marescaux, D.Verkest, J-Y.Mignolet, and S.Vernalde. perating-system controlled network on chip. In 41*th* Acm/Ieee *Design Automation Conf.*, USA, 2004.

[6] E. D. Jensen, J. D. Northcutt, "Alpha: a nonproprietary OS for large, complex, distributed real-time systems," in *Proc. IEEE Workshop on Experimental Distributed Systems*, Huntsville, Alabama, pp. 35-41, October 1990.
P. Gerum, "Xenomai - Implementing a RTOS emulation framework on GNU/Linux," Whitepaper, 2004.

[7] Z. Chen , X. Luo , Z. Zhang, "Research Reform on Embedded Linux's Hard Real-time Capability in Application," Embedded Software and Systems Symposia, 2008. ICESS Symposia '08. International Conference on 29-31 July 2008 Page(s): 146 - 151.

[8] EMBEDDED.COM, http://www.embedded.com.

[9] http://en.wikipedia.org/wiki/eCos

[10] http://en.wikipedia.org/wiki/Soft_error

[11] http://en.wikipedia.org/wiki/Real-time_operating_system

[12] http://en.wikipedia.org/wiki/System-on-a-chip

Swapan Debbarma completed his B.Tech. degree in Computer Science & Engineering from NERIST, AP and M.Tech.from TU, Tripura-Agartala, India, in the same branch. Mr. Debbarma is working in NIT, Agartala as an Asst. Prof in CSE Department since 2001 and persuing his Ph.D in from NIT, Agartala. His field of interests are Image processing, OS, Networking etc.