

A Reliable Task Scheduling Scheme For Sensor-based Real-time Operating System

YINGWU WANG, XIAOHUA LUO, KOUGEN ZHENG, ZHAOHUI WU, YUNHE PAN

College of Computer Science
Zhejiang University
Hangzhou, Zhejiang Province 310027, China

Abstract: - The emergence of wireless networked sensors constitutes a hot research topic in embedded system design. Although system resource is seriously limited, operating system applied in sensors has to implement complex task scheduling, which should support concurrent operations, real-time constraint, adaptability and reliability. In this paper, we present a reliable OS scheduling scheme for wireless networked sensors. According to the analysis of operating model and task set of wireless networked sensors, a primary and subordinate two-level task-scheduling scheme is implemented to schedule time-critical and non-time-critical tasks, and an admission control policy is employed to meet hard real-time requirement of some real-time tasks. Furthermore, based on this scheme, some fault tolerance strategies are outlined to improve reliability at run time, including fault tolerance analysis, admission control based on fault tolerance, fault detection, and fault recovery.

Key-Words: - scheduling scheme, real-time, wireless network sensor, fault tolerance

1 Introduction

The capability to sense environment is a critical element of pervasive computing [1]. Embedded objects that support pervasive computing are more and more equipped with computational power that allows them to be smart devices with the ability to interact with their environment. One of the most challenging research fields is wireless networked sensors.

A typical wireless networked sensor is usually made up of four basic components: a sensing/actuating unit, a processing unit, a communicating unit and a power supply unit. These sensors have two remarkable characters: limited in size and autonomous in operation. In spite of the resource limitation, the tiny sensors have to execute complicated autonomous operation, and it should meet the requirements of concurrent operation, real-time constraint, and self-adaptability. And also, since these tiny devices are usually numerous and largely unattended, they will be expected to be operational for a large fraction of the time without human intervention. Furthermore, error recovery is often too complex to apply in sensors, there is no real recovery mechanism except for automatic reboot, and great efforts should be taken to improve their fault tolerance and reliability. Small physical size, modest active and power load are provided by the hardware design. And the concurrency-intensive operation and real-time constraints should be implemented by software. A tiny operating system is needed, which not only

retains these characteristics by managing the software/hardware effectively, but also provides a scheduling scheme to achieve efficient reliability and robustness. Reliability can also be achieved through redundancy. However, due to the seriously limitation in weight, space, power and cost, traditional hardware redundancy techniques can not be applied in wireless networked sensors. Thus, it is essential for software to enhance the reliability of individual devices.

OS scheduling scheme in traditional embedded systems are well-studying today. However, in small, low-power and embedded wireless networked sensors, it is quite a different novel region to study. To meet this requirement, we present an embedded operating system γ OS, in which we concepts and provides a flexible and reliable scheduling scheme for tasks with different time constraint.

2 Related Work

A large amount of work has been completed on developing wireless networked sensors, especially the operating system for tiny sensors [2, 3, 4, 5]. Creem[2] and pOSEK[3] are typical real-time operating system designed for deeply embedded systems, but their control-centric design is very different from dataflow-centric design in networked sensors.

TinyOS [4], with its component-based architecture, tasks and event-based concurrency and split-phase

operations, is applicable for dataflow-centric applications. However, it simply uses a LCFS events scheduler and a non-preemptive FIFO tasks scheduler to manage sensor operation. Adaptability and fault tolerance has not been considered. MANTIS OS [5] resembles classical, UNIX-style schedulers. However, it still does not consider fault tolerance for sensors adequately.

As we mentioned above, reliability is very important for wireless networked sensors. To improve reliability, many scheduling algorithms have been proposed to achieve executing efficiency and fault tolerance in real-time embedded systems. Generally, some typical scheduling algorithms include three broad categories [6]: static scheduling, dynamic scheduling, and scheduling of imprecise computations.

A static scheduling is calculated off-line and fixed for the life of the system, its adaptability is poor. A dynamic scheduling makes its scheduling decisions at run time based on requests for system services, such as RMS (rate-monotonic scheduling) [7] or EDF (earliest deadline first) [8]. Although microprocessor in sensors is seriously restricted in computation power, it is necessary to execute complex tasks with precedence, synchronization, and exclusion constraints under certain condition. It is still difficult to not only assure the schedulability of tasks, but also meet the requirement of restricted-resource. To meet these requirements, γ OS presents a new task scheduler, which applies the RMS algorithm to schedule time-critical tasks and adopts an admission control policy to meet their real-time requirements, and fault tolerance based on this scheme is also considered.

3 Task Model Analysis

The emergence of wireless networked sensors has created a wide space of new problems in systems design. γ OS is an embedded operating system developed specially for these tiny devices. Therefore, it should meet the special application requirements. Often, the processor and other resources used in such applications are shared by a certain number of time-critical monitoring and control functions and a number of non-time-critical jobs.

We define a task as a basic logic unit of programming that an operating system controls. It is a schedulable entity which can be appointed with priority and will compete with other concurrent tasks for processor execution time. Figure 1 shows the sensor task model.

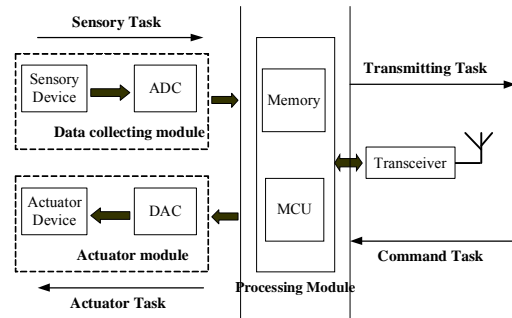


Fig.1 sensor task model

In γ OS, Tasks can be classified into four types:

- 1. Sensory task T_s denotes the task that collects information from environment, and processes the collected information. Task set $T_s = \{ T_{s1}, T_{s2}, \dots, T_{sn} \}$,
- 1. Actuator task T_a denotes the task that generates control signals for the actuator devices and operates the actuator devices. Task set $T_a = \{ T_{a1}, T_{a2}, \dots, T_{an} \}$
- 1. Command task T_c denotes the task that takes actions according to the commands from other sensor nodes or central machine. Task set $T_c = \{ T_{c1}, T_{c2}, \dots, T_{cn} \}$
- 1. Transmitting task T_t denotes the task that transmits data by transceiver, including publishing the services and status information, or sending required data to subscribers. Task set $T_t = \{ T_{t1}, T_{t2}, \dots, T_{tn} \}$

The whole task set $T = \{ T_s, T_a, T_c, T_t \}$.

Table 1 Analysis of the task set of wireless sensors

Task	T_a	T_s	T_c	T_t
Activation	Periodic	Periodic	Sporadic	Sporadic
Priority	High	High	Low	Low
Deadline	Actuating interval	Collecting interval	None	None
WCET	$t_a + t_{DA}$	$t_{AD} + t_s + t_p$	$t_p + t_i$	$t_p + t_i$
Resource conflict	Actuator device, Processor, DAC	Sensory device, Processor, ADC	Processor, Transceiver	Processor, Transceiver
Triggered by	Timely interrupt	Timely interrupt	Command interrupt	Posted by T_s or T_c

t_{DA} and t_{AD} denotes D/A or A/D converting time, t_s denotes sensing time, t_a denotes actuating time, t_p denotes data processing time, t_i denotes data transmitting time

Table 1 shows the property analysis of the task set. Task T_s collects and processes information from environment. It is periodic and triggered by timer. The deadline of T_s is usually the periods of collecting operations. Task T_a generates control signals and operates the actuator devices according to the processed data from T_s , or the command from T_c . The deadline of T_a is usually the periods of activating operations. For example, in a temperature control system, if the sensory devices of the sensors detect

that the environment temperature exceeds a certain bound, the actuators of the sensors must be activated in time to impact the temperature of the environment. Task T_s and T_a are time-critical, and all the T_s and T_a tasks have higher priority than the T_c and T_t tasks. The priority level of task T_{a_i} is higher than that of T_{s_i} , for the task T_{a_i} is usually more critical than T_{s_i} , in the case that it is activated.

Task T_c responds the request or command from other sensor nodes or central machine, and processes the received data from other sensor nodes. Task T_c is activated when the sensor receives command from other sensor nodes or central machine. Task T_t publishes the service and status information of the sensor node, transmits the requested data to other sensor nodes. Task T_t is posted by T_s when the system status has changed or by T_c when sensor data are requested by other devices. Task T_c and T_t are non-time-critical, and all the T_c and T_t tasks have lower priority than the tasks T_s and T_a . The priority level of task T_{c_i} is equal to that of T_{t_i} .

4 Scheduling Scheme Implementation

4.1 Basic Analysis

According to the above analysis, the scheduling scheme and priority policies for γ OS should meet the following requirements:

- 1 The scheduler should support periodic tasks and sporadic tasks scheduling.
- 1 T_a and T_s tasks are time-critical and should meet their real-time constraint.
- 1 T_c and T_t tasks are non-time-critical.
- 1 The task priority level is: $T_a > T_s > T_t = T_c$.
- 1 The priority levels of T_s tasks are based on their periods. The T_s with shorter period is appointed with higher priorities.
- 1 The priority levels of T_a tasks are based on their periods. The T_a with shorter period is appointed with higher priorities.
- 1 Let $T_{s_1}, T_{s_2}, \dots, T_{s_m}$ denote a set of priority-ordered tasks with T_{s_m} being the task with the lowest priority.
- 1 Let $T_{a_1}, T_{a_2}, \dots, T_{a_m}$ denote a set of priority-ordered tasks with T_{a_m} being the task with the lowest priority.
- 1 Task T_s with data processing commonly more complex than task T_a , so we can assume the execution time of T_a task is shorter than that of T_s task.
- 1 Admission control policy should assure the schedulability of task set when it is changed by the arriving of new task.





4.2 Scheduling Scheme

γ OS scheduler provides two-level task scheduling scheme, which includes primary task scheduling and subordinate task scheduling, to isolate time-critical tasks and non-time-critical tasks.

The primary task scheduling is responsible for scheduling the task T_a and T_s , and a background task T_b . The primary task scheduling must meet the requirement of time constraint of task T_a and T_s . In the primary task scheduling, task T_b is imported to isolate time-critical tasks and non-time-critical tasks. T_b possesses the lowest priority level. When task T_b is scheduled, the processor switches to subordinate task scheduling. There are many existing scheduling schemes [9] which can be adapted to primary task scheduling. Based on the analysis in section 4.1, RM scheduling algorithm [7] is adapted.

The subordinate task scheduling is responsible for scheduling the lower priority task T_c and T_t . T_c and T_t tasks are non-time-critical. T_{c_i} and T_{t_i} have equal priorities and can not preempt each other. T_c and T_t can be preempted by time-critical task T_a and T_s . The "First Come First Served" (FCFS) policy is adapted.

Table 2. An instance of task set

Task	Priority	Period	CPU time	Legend
T_a	High	6	1	
T_s	Med High	8	2	
T_c	Low	Sporadic	2	
T_t	Low	Sporadic	5	

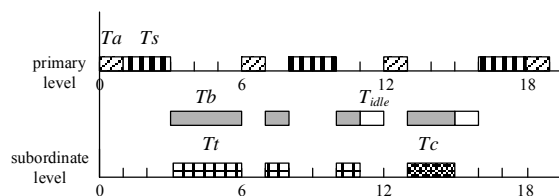


Figure 2. Basic scheduling of a task set

Table 2 listed an instance of task set which contains four tasks: one T_a task, one T_s task, one T_c task and one T_t . The priority level of every task is assigned as: $T_a > T_s > T_c = T_t$. Figure 2 shows how the instance is scheduled. The blank section task T_{idle} indicates idle of the processor. If T_{idle} is scheduled, the processor will enter a sleeping mode, which will greatly reduce power consumption of sensors. Using the two-level task scheduling policy, time-critical task set T_a and T_s can be efficiently distinguished from non-time-critical task set T_c and T_t . Furthermore, priority ceiling protocol [11] is adapted to avoid the problem of priority inversion and deadlock.

The impact of interrupt service routines (ISR) is also needed to be considered, for the ISR may expend

much CPU time in resource restricted sensors. For instance, three types of interrupts are considered in γ OS. There are two timer interrupts It_a and It_s , It_a triggers task Ta timely, and It_s triggers task Ts timely. The second is command arriving interrupt Ic , which indicates that a new command has arrived. The third is transmission completion interrupt It , which indicates that the data transmission has been completed. Furthermore, The A/D and D/A conversion interrupt are also considered. I_{AD} indicates task Ts that the AD conversion has been completed, and I_{DA} indicates task Ta that the DA conversion has been completed.

4.3 Admission control policy

When a new task is triggered, the status of whole task set is changed. The schedulability of the whole task set must be discussed, especially for the real-time task set. In γ OS, we mainly discuss the schedulability of the real-time task set. An admission control policy is applied to assure the schedulability of the changed task set. The tasks in primary task scheduling level are hard real-time, they must be considered under admission control policy administration. In the basic RMS, Liu and Layland [7] have derived a simple schedulability test based on the resource utilization by the task as shown in (1).

$$U \leq n(2^{1/n} - 1) \quad (1)$$

For the task set we described in section 3, Ta task set can be denoted as $Ta_1 (Ca_1, Pa_1), \dots, Ta_n (Ca_n, Pa_n)$, where Ca_i denotes the execution time of task Ta_i , Pa_i denotes the period of task Ta_i , and $Pa_i \leq Pa_{i+1}$, $i=1,2,\dots,n-1$. Ts task set can also be denoted as $Ts_1 (Cs_1, Ps_1), \dots, Ts_n (Cs_n, Ps_n)$. In terms of the task in primary scheduling level, the utilization factor is as shown in (2).

$$U = \sum_{i=1}^n Cs_i / Ps_i + \sum_{i=1}^n Ca_i / Pa_i \quad (2)$$

Task Tb is a background task without deadline, so it is not considered in schedulability test. The shortcoming of the equation (2) is that it does not take the effect of the resource access control protocol and interrupt service routines (ISR) into account. According to the result proved by Sha, Rajkumar and Lehoczky [10], the primary task scheduling using the priority ceiling protocol can be scheduled by the rate-monotonic algorithm if the following condition is satisfied:

$$\sum_{i=1}^n Cs_i / Ps_i + \sum_{i=1}^n Ca_i / Pa_i + \max \left\{ \frac{Bs_1}{Ps_1}, \mathbf{L}, \frac{Bs_n}{Ps_n}, \frac{Ba_1}{Pa_1}, \mathbf{L}, \frac{Ba_n}{Pa_n} \right\} \leq n \left(2^{1/n} - 1 \right) \quad (3)$$

In inequation (3), Bs_i and Ba_i separately denotes the worst case blocking time of task Ts_i and Ta_i . The value of Bs_i and Ba_i are determined by the scheduling scheme and resource access policy. Considering that the priority ceiling protocol is adopted, Bs_i or Ba_i is the worst access time of the resource possessed by lower priority task. In γ OS, the worst access time for all the resources, can be determined approximately, let B_w denotes it. Therefore, we can simplify the calculation in a pessimistic way shown in (4):

$$\sum_{i=1}^n Cs_i / Ps_i + \sum_{i=1}^n Ca_i / Pa_i + \frac{B_w}{P_{min}} \leq n(2^{1/n} - 1) \quad (4)$$

In inequation (4), P_{min} is the minimal interval of all the Ta and Tc tasks.

Now, we consider the effect of ISRs. Let C_{Ia} , C_{Is} , C_{Ic} and C_{It} denote the worst ISR execution times of interrupt It_a , It_s , Ic and It . P_{Ia} , P_{Is} , P_{Ic} and P_{It} denote the minimum triggered intervals. So the admission control algorithm of γ OS can be adjusted as:

$$\sum_{i=1}^n Cs_i / Ps_i + \sum_{i=1}^n Ca_i / Pa_i + \frac{B_w}{P_{min}} + \frac{C_{Ia}}{P_{Ia}} + \frac{C_{Is}}{P_{Is}} + \frac{C_{Ic}}{P_{Ic}} + \frac{C_{It}}{P_{It}} \leq n(2^{1/n} - 1) \quad (5)$$

Inequation (5) is too pessimistic for most cases. Whether for Ta task or for Ts task, the actual executing time is greatly less than WCET in most cases. For example, in temperature control sensor system, Ta task is activated only when the environment temperature exceeds a certain bound. The time saved by short Ta and Tc task, named *slack*, can be assigned to the background task Tb , which ensures Tt and Tc tasks could be executed. No admission control policy is currently considered for the subordinate task scheduling as the tasks Tt and Tc are not time-critical.

5 Fault Tolerance Implementation

Fault tolerance is increasingly important in modern autonomous sensor system. In a typical deployment, hundreds of sensors will be expected to work unattendedly for a long period. Though sensors are resource restricted, fault tolerance should be considered enough.

Generally, there are three kinds of faults: permanent, intermittent, and transient faults. Most permanent faults are due to the breakdown of computing unit. Usually, they are resolved by hardware redundancy. Transient faults are hard to detect, but significantly more frequent than permanent faults [11]. The incorrect data race, unexpected temporary hardware errors can be the cause. Intermittent faults are nightmare for micro embedded sensors. But

sometimes, it is the result of repeated transient faults. Therefore, γ OS pay attention on the tolerance of transient faults by adding time redundancy to reexecute the wrong task. More research has been conducted on detecting and tolerating faults using both hardware [12] and software [13].

5.1 Fault Tolerance Analysis

Since γ OS focuses on the implementation of transient fault tolerance by reexecuting the wrong task, two attribute requirements should be considered in the task model for fault tolerance.

- 1 Time redundancy: The reserving time should be adequate to reexecute the task. This condition can be converted into a scheduling problem where the schedule scheme should guarantee error recovery from transient faults by providing sufficient backup time for re-execution [14].
- 1 Data integrity: The shared variables and status information should be backed up to reexecute the task. Each task instance maintains two copies of status information and variables, named primary/backup data block, to support fault tolerance.

When a task is firstly invoked, the two blocks are initialized with the same data. The primary block is updated synchronously according to the task executing. If the task is finished successfully, the primary block will contain information that indicates the task is finished successfully, and next task can be scheduled. If not so, the γ OS will reexecute the task based on the backup data block. For example, if a task fails to operate actuator device at first time, it can be re-scheduled till it executes successfully.

Once a new task arrives, the admission control starts to test whether this new task can be dispatched, and decide the schedule strategy according to the result of evaluation. In a general way, the typical RMS is adapted to schedule active tasks in the primary level. When a task is terminated, a fault detection mechanism detects whether a fault occurred during the task execution. If time-critical tasks, such as T_s or T_a , are terminated with errors, scheduler would reexecute them under certain fault recovery scheme. For T_c and T_t are not time-critical, they can be re-execution more freely.

5.2 Admission control

The system needs to reserve some extra utilization for the re-execution to tolerate transient fault. The tasks can be guaranteed to meet their deadlines under an assumption of minimum fault interval if the following condition is satisfied [15].

$$\sum_{i=1}^n (Uc_i + Ua_i) \leq U_{LL} (1 - U_{max}) \quad (6)$$

In inequation (6), U_{LL} denotes the Liu and Layland bound, and U_{max} is the backup utilization of task ($U_{max} = \max\{U_i\}$) to protect T_a and T_s task from faults.

Based on the discussion in section 4.3, the admission control policy can be improved to the following condition.

$$\sum_{i=1}^n C_s/P_s + \sum_{i=1}^n C_q/P_q + \frac{B_w}{P_{min}} + \frac{C_{Ia}}{P_{Ia}} + \frac{C_{Is}}{P_{Is}} + \frac{C_{Ic}}{P_{Ic}} + \frac{C_{It}}{P_{It}} \leq U_{LL} \left(1 - U_{max} - \frac{B_w}{P_{min}} \right) \quad (7)$$

For tasks T_t and T_c are non-time-critical, Admission control policy in the subordinate task scheduling level can be achieved by using buffer queue. New T_t and T_c tasks can be queued to wait for the assignment of processor, I/O or communication resources.

5.3 Fault Detection

To provide fault tolerant capabilities, the methods of detecting failures must be implemented first. A task status structure is added to record task information. When a fault occurs, error information will be recorded in task status structure. An exception handler, a safety monitor, and even the task itself, can complete this operation. After the task procedure is terminated, the task status structure should be checked to estimate whether a fault occurs. If so, the scheduler then decides how to deal with it, simply abandon it or reexecute it.

Actually, wireless networked sensors are resource restricted devices used to monitor and control environment. In terms of non-time-critical tasks, a better choice is to abandon them and wait for the next same type task.

5.4 Fault Recovery

Once a task terminates and fault information is found in task status structure, γ OS scheduler will evaluate the task. There are three strategies can be employed:

- 1 Simply abandon, such as some T_c and T_t tasks that can not be accomplished in time limit. They will be invalid in the next period.
- 1 Reexecute with some delay, including some backup T_c or T_t tasks. They are crucial, but not time-critical. Once a fault occurs, they can be reexecuted with its original subordinate priority.
- 1 Reexecute immediately, such as some T_a or T_s tasks that are crucial and time-critical. To finish them within time limit, scheduler will reinitialize and reactivate the task according to backup data block, and reexecute it immediately.

The third kind of operation is the most complicated. In section 4.3 we have mentioned *slack* saved by short task T_a and T_s . Fault recovery can be achieved by adding time redundancy. One general approach of fault-tolerance is to make sure that there is enough *slack* time to allow for the re-execution of fault task. Slacks are distributed throughout the schedule in the form of backup slots at each period boundary. [15] showed that the following conditions must hold true to ensure re-execution of any one faulty task:

- 1 For every task T_i , a slack of at least C_i should be present between kT_i and $(k+1)T_i$.
- 1 If there is a fault during the execution of task T_i , the recovery scheme should enable task T_i to reexecute for a duration C_i before its deadline.
- 1 When a task reexecutes, it should not cause any task to miss its deadline.

Let U_B denotes the backup utilization for any task. [15] has proved that if $U_B = \max\{U_i\}$ and recovery scheme RS is adopted then the above conditions will be satisfied. The recovery scheme RS is that any instance of a task that has a priority higher than that of task T_i and a deadline greater than D_i (deadline of task T_i) will be delayed until recovery is complete.

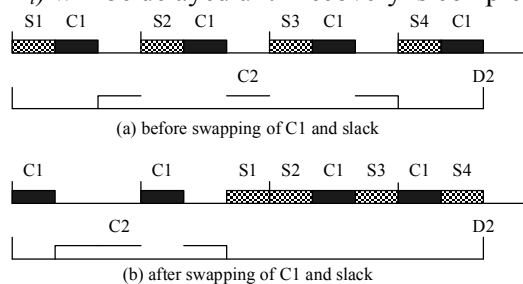


Fig. 3. An instance of fault recovery by swapping slack

Fig. 3 shows an example of fault recovery applying RS on two tasks T_1 and T_2 , in which C_1 and C_2 are the execution time of T_1 and T_2 separately, D_2 is the deadline of task C_2 , S_i is the slack time of each period boundary. This figure illustrates that if T_2 fails then it can be re-executed using the swapped slack before deadline.

6 Conclusion

In this paper, we proposed a task scheduling scheme for sensor-base real-time system. After analyzing the operating model and task set, a two-level task scheduling scheme is implemented in the scheduler of γOS . RMS algorithm is used to schedule time-critical tasks, and an admission control policy is applied to meet real-time requirement. Furthermore, some fault tolerance strategies are also taken into account to improve reliability at run time, including

fault tolerance analysis, admission control based on RMS, fault detection, and fault recovery.

References:

- [1] M. Weiser, The Computer for the 21st Century, Sci.Amer., Sept. 1991.
- [2] <http://www.goofee.com/creem.htm>.
- [3] <http://www.isi.com/products/posek/index.htm>.
- [4] <http://webs.cs.berkeley.edu/tos/>
- [5] <http://mantis.cs.colorado.edu>
- [6] N. Kandasamy, J. P. Hayes, and B. T. Murray, "Scheduling Algorithms for Fault Tolerance in Real-Time Embedded Systems." Dependable Network Computing, D. Avresky (Ed.), Kluwer Academic Publishers, Boston, 1999.
- [7] C. L. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," J. Assoc. Comput. Mach., vol. 24, 1973, pp. 46-61.
- [8] J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo, "Deadline Scheduling For Real-Time Systems: EDF and Related Algorithms." Kluwer Academic Publishers, 1998.
- [9] D. B. Stewart and P. K. Khosla, "Real-Time Scheduling of Sensor-Based Control Systems", in Proceedings of Eighth IEEE Workshop on Real-Time Operating Systems and Software, Atlanta GA, May 1991, pp. 144-150.
- [10] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization." IEEE Trans. Computers, 1990.
- [11] X. Castillo, S.R. McConnel, and D.P. Siewiorek, "Derivation and Calibration of a Transient Error Reliability Model." IEEE Trans. on Computers, 1982, pp. 658-671.
- [12] J. Gaisler, "Concurrent Error-detection and Modular Fault-tolerance in a 32-bit Processing Core for Embedded Space Flight Applications". In Symp. on Fault Tolerant Computing (FTCS-24), pages 128-130. IEEE, 1994.
- [13] B. Randell. System Structure for Software Fault Tolerance. IEEE Trans. on Software Engineering, SE-1(2):220-232, June 1975.
- [14] Dong, Libin (ed.), "Implementation of a Transient-Fault-Tolerance Scheme on DEOS," In Proceedings of the Real-Time Technology and Application Symposium, Vancouver, Canada, 1999.
- [15] S. Ghosh, R. Melhem, D. Mosse, and J. Sen Sarma, "Fault Tolerant Rate Monotonic Scheduling." Journal of Real-Time Systems, vol.15, no.2, 1998, pp.149-181.