Real-Time Scheduling Algorithms

Vimal Kumar
Computer Science and Engineering Department
The University of Texas at Arlington, Arlington, TX 76019
vkumar@cse.uta.edu

Abstract: *Real-time systems play an important role in our modern society. They are used in critical control systems that rely on timely response and computed results to function properly. This paper summarizes to an extent the state of real-time systems in the area of scheduling. A vast amount of work that has been done in the area of real-time scheduling by both the operations research and the computer science communities. While concerning ourselves with the scheduling requirements and methodologies of a real-time application, we observe that it is fundamentally different from a non time-constrained time-sharing application. The difference arises in the methods used for scheduling the tasks and the system support that is necessary for such an implementation. In this paper, we will consider the various issues that typify real-time scheduling. We discuss two major algorithms that forms a baseline for all scheduling approaches and we present a real-time implementation of such a system.*

Real-time systems differ from non-real-time systems in that they react to events of the physical world within a certain duration of time. Such a real-time system monitors and controls external processes, and must react to changes in a timely fashion, sometimes in the order of milliseconds. For simple enough processes the response is quick enough, but for many real-time systems that are more complex, a sophisticated coordination is required and rapid response to events becomes challenging. This motivates the study of methods to schedule various real-time events[1] so that time critical events are processed and responses are available to these real-time systems. These responses are critical because failure to respond to such events might result in failure of the system leading to loss of life and property.

Two important properties of a task are the nature of the deadline and the arrival characteristics. The task arrival may be periodic with a constant interval, or the task may be stastically distributed and the inter-arrival time may be random in nature. The important requirement is that a time constrained task should be completed before the deadline set for that task expires. Depending on the real-time system, the nature of the deadline maybe different. Tasks with hard deadlines must be completed before the specified time. If the deadline is missed, the utility of execution for such a task in nil. A hard deadline[1] that is missed results in a failure of the system, depending on the task. On the other hand, tasks that gradually degrade with the deadline are called soft deadlines[16]. Such tasks impose a more flexible restriction on the scheduler and missing deadlines will not cause a catastrophic failure[1].

In this paper, we are concerned with abstract timing requirements in a hard-real-time environment, and the different ways of mapping these requirements to tasks with timing constraints. The methods for scheduling based on the task, and the mechanisms for scheduling such tasks in a real-time kernel are discussed. In Section 2 we discuss the classic scheduling algorithms in a multiprogramming system. Section 3 gives us additional information about the nature of real-time scheduling and some of the problems associated with priority inversions, Section 4 delineates some of the solutions to resource reclaiming which is shown to improve the utilization of the CPU. Section 5 introduces us to the concept of designing compilers for real-time systems with some introduction to the task ordering that is carried out in such applications. In Section 6 we discuss the implementation of a real-time system; Mach. We finally conclude with discussion in Section 7 and Section 8.

2. Scheduling Algorithms:
Early work was carried out by Liu and Layland[2] who presented scheduling algorithms for fixed and dynamic tasks. The rate monotonic algorithm was shown to be useful for fixed priority tasks, and the earliest-deadline-first and minimum laxity first algorithms was proved to be useful for dynamically changing tasks. This gives us a broad classification of scheduling algorithms namely: *static priority* [2] and *dynamic priority scheduling*[2]. In all the related work that followed, the above algorithms were the basis for further optimization.

2.1 Rate Monotonic Algorithm (RM)
This is a fixed priority algorithm[2] and follows the philosophy that higher priority is given to tasks with the higher frequencies. Likewise, the lower priority is assigned to tasks with the lower frequencies. The scheduler at any time always choose the highest priority task for execution. By approximating to a reliable degree the execution times and the time that it takes for system handling functions, the behavior of the system can be determined apriori.

The rate monotonic algorithm[2, 7, 9] can successfully schedule tasks in a static priority environment but it has bound of less that 100% efficiency. The CPU utilization of tasks τ_i where $1 \leq i \leq n$, is computed as the ratio of worst case computing time C_i to the time period T_i. The total utilization of the CPU is computed as follows[2]:

$$U_n = \sum_{i=1}^{n} \frac{Ci}{Ti} \qquad (1)$$

Here the frequency of the task is the reciprocal of the time period of the particular task. For the RM algorithm the worst-case **schedulable time bound** W_n for a set of n tasks was shown to be [Liu]:

$$W_n = n(2^{1/n} - 1) \qquad (2)$$

From(2), we can observe that $W_1 = 100\%$, $W_2 = 83\%$, $W_3 = 78\%$ and as the task set grow in size, $W_n = 69\%$ (ln2). Thus for a set of tasks for which the total CPU utilization is less than 69% means that all the deadlines will be met. The tasks are guaranteed to meet their deadlines if $U_n \leq W_n$. If Un > Wn, then only a subset of the original task set can be guaranteed to meet the deadline which forms the upper echelon of the priority ordering. This set of tasks will be the critical set[7].

Another problem that exists is the inability for RM to support dynamic changes in periods, which is a regular feature of dynamically configurable systems. For example, consider a task set of three τ_1, τ_2, and τ_3, with time periods T_1=30ms, T_2=50ms and T_3=100ms respectively. The priorities assigned are according to the frequency of occurrence of these tasks and so τ_1 is the highest priority task. If the period for the first task changes to T_1=75ms, we would then under RM require that the priority orderings be changed to, $\tau_2, \tau_1,$ and $\tau_3.$ This change is detrimental to the completion of the scheduled jobs, who have to finish before their deadlines expire. The problem with RM encouraged the use of dynamic priority algorithms[9].

2.2 Earliest Deadline First Algorithm(EDF)

This algorithm takes the approach that tasks with the closest deadlines should be meted out the highest priorities. Likewise, the task with the latest deadline has the lowest priority. Due to this approach, the processor idle time is zero and so 100% utilization can be achieved. The necessary and sufficient condition for the schedulability of the tasks follows that [2, 9]: for a given set of n tasks, $\tau_1, \tau_2 ... \tau_n$ with time periods T_1, T_2… T_n, and computation times of C_1, C_2… C_n, the deadline driven schedule algorithm is feasible if and only if [2]

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots. + \frac{C_n}{T_n} \le 1 \qquad (3)$$

The limitation of the EDF is that we cannot tell which tasks will fail during a transient overload. Even though the average case CPU utilization is less than 100%, it is possible for the worst-case utilization to go beyond and thereby the possibility of a task or two being aborted. It is desirable to have a control over which tasks fail and which do not, however, this is not possible in EDF. The situation is somewhat better in RM because it is the low priority tasks that are preempted [9].

3. Scheduling Real-Time tasks

The nature of real-time systems presents us with the job of scheduling tasks that have to be invoked repeatedly. These tasks however may range from simple periodic tasks with fixed execution times to dynamically changing aperiodic tasks that have variable execution times. Periodic tasks are commonly found in applications such as avionics and process control requiring data sampling on a continual basis. On the other hand, sporadic tasks are associated with event-driven processing such as user response and non-periodic devices. Given a real-time system the goal of a good scheduler is to schedule the system's tasks on a processor, so that every task is completed before the expiration of the task deadline. These and some of the other issues like stability and transient overloads are examined here.

3.1 Cyclic Scheduling

The cyclic schedule is a timed sequence of predetermined tasks that are repeated indefinitely in a cyclic fashion. The cyclic processing [10] is designed for environments that resemble periodic processes and background processing. The scheduler can schedule other aperiodic tasks when the CPU is not servicing periodic tasks. Deterministic scheduling may be used to find schedules that meet the timing constraints of all the tasks. However, to obtain an optimal scheduling algorithm apriori knowledge of the timing parameters of all tasks is required, and optimal non-preemptive scheduling of computations with given timing constraints is NP-hard.

3.2 Sporadic Tasks
There can be many periodic tasks [10] whose periods can be arbitrary. The arrivals may be bursty too. Scheduling can be achieved in several ways.

- We could find a way to allow for the sporadic tasks to preempt periodic tasks keeping in mind that deadlines are not missed.
- Schedule the tasks with a maximum possible execution rate.
- Schedule a periodic task that completes sporadic tasks that are queued for servicing.

3.3 Transient Overloads
In many applications the execution times are not fixed and are stochastic. It becomes supremely important to deal with the completion of tasks that are critical to the mission, even at the cost of missing the deadlines of a few non-critical tasks.

A scheduling algorithm is considered stable if there is a fixed set of tasks that meet their deadlines even during transient conditions. To ensure that the critical set tasks meet their deadlines they have to belong to the critical stable set. The RM algorithm is a stable scheduling algorithm because it guarantees that tasks with the highest priorities will always execute.

Rarely, we come across a task with a longer period that could be critical to an application but does not belong to the stable set of the RM algorithm. This problem is solved by the period transformation technique, which ensures high utilization of the CPU while ensuring deadlines of important tasks. Period transformation involves the splitting of a long-period important task over several short periods. We increase the frequency of these tasks by giving it a period $T_i/2$ and suspending it after it executes half its worst time execution C_i / 2. A procedure for period transformation with minimum task portioning is found in [17].

Periodic transformations allow for important tasks to have high priority while maintaining consistent RM rules. These kinds of transformations are familiar in a cyclic executive environment. The advantage of this is that we do not have to have variable code segments to fit shared time slots. Instead, τ_i is simply suspended after performing a portion of the work.

Another advantage of the transformation being that, integral multiple transformation periods allow for a higher utilization of the CPU for the RM algorithm. The periods cannot get too short, otherwise the scheduling overhead adds to delays[10].

3.4 Controlling Priority Inversions and Critical Regions
In real-time systems, a task priority indicates the importance that the programmer places on each task. Because of systems that should respond to changing events, and so changing priorities, the scheduler must be able to react to a changing task mix. A situation can arise, where the scheduler may not be able to respond immediately. For example, it is often necessary to disable hardware interrupts and during such times the system is oblivious to external events. This part of code where the interrupts are disabled is called the critical region [11].

3.4.1 Priority Inversions
During execution of a task in the critical region, the priority structure is suspended and the executing task essentially transforms itself into the highest priority task. This can cause havoc in the priority ordering of the system. The problem arises, when a low priority task is in its critical

region and a high priority task becomes ready to run. This does not cause an interrupt and so the high priority task is delayed while the low priority task is executed.

Critical regions in user applications are protected by semaphores or other synchronization mechanisms and are crucial to systems using shared memory. These have a small impact on the performance of the system but the non-preemptable nature has many implications. Reducing the adverse effects of such tasks is the focus of research to reduce or eliminate critical sections, avoid priority inversions and improve response times [11, 10].

4. Other Scheduling Issues

In this section we discuss two issues which are important to a real-time system namely: fault tolerance and improving performance of CPU utilization. A third issue concerns the scheduling of imprecise computations, which is a trade-off between a correct computation versus the timeliness of the delivered results [5].

4.1 Scheduling With Fault Tolerance Constraints

In a proposal, a deadline mechanism is detailed that can guarantee that a primary task can meet its deadline if there are no failures, and if there is a failure, an alternative task will run before the deadline. These are allowed only for periodic tasks that can be preempted. It is possible to compute a tree of schedules and the tree can be encoded within a scheduler.

Specifically, we are trying to switch to a new task upon failure of a mission critical task, where the new schedule has been precomputed. These dynamic schedules are computed off-line to arrive at the final set of contingency schedules. These schedules are embedded within the primary schedule to ensure that hard deadlines are met in the event of a failure. The processing time will increase during such schedules switchover but that is the price of having little or no time to respond to failures during task execution.

Such approaches are very important for static, embedded computer systems where fault tolerance is important and deadlines are tight. Such systems do not optimize on processor utilization, but the task completions guarantees are of significance. However, while dealing with dynamic environments of next generation real-time systems, we need to derive applications that can accept a trade-off between fault-tolerance and timeliness. Planning becomes possible with the use of dynamic planning-based schedulers where an adaptive fault-tolerant approach is taken to permit the forecasting of timing errors, and thus helps in graceful degradation [5].

4.2 Scheduling With Resource Reclaiming

Task execution times may vary and as a result we might have a few tasks completing earlier than expected by the scheduler. The extra time that becomes available can be reclaimed by the task dispatcher and utilize this time to execute other tasks. This approach can be utilized to execute non-real-time tasks during such idle time slots. But, importantly the tasks with timing constraints should be ensured of completion. We need to take care of such facilities in a multiprocessor environment where we need to conserve precedence constraints on tasks to preserve the task ordering. Task replicas (for fault tolerance) should follow a consistent schedule even when the tasks are run in parallel.

Reclaiming unused time in systems that do dynamic planning must be correct, i.e., they must maintain the feasibility of guaranteed tasks. The overhead of resource reclaiming should be less compared to the tasks computation times. Moreover, these should have bounded complexity, in that; the algorithm should be independent of the number of tasks in the schedule.

Mostly two kinds of resource reclaiming approaches are taken; Basic Reclaiming and Reclaiming with an early start. These form the strategies for dynamic load optimization of a multiprocessor schedule. Both these approaches have a bounded time complexity [5].

5. Compiler Designs for Real-Time Programs:
Two aspects of real-time systems are the functional requirements and temporal requirements. Functional requirements specify the valid transitions that gives us a valid output upon providing a valid input. Temporal requirements on the other hand, give us an idea of the bounds of occurrences of events. For example: the time bounds for the movements of a robot arm. The action should be initiated and completed within a timebound. A balance between these two requirements can be struck with the help of specialized programming languages like TCEL(Time-constrained Event Languages)[8], that combine the timing relationship between observable events with timing constructs and semantics built into the low level code[8].

5.1 Time-Constrained Event Languages
With TCEL, code-based semantics is converted to event-based semantics, thus the task of ensuring timing constraints of code blocks need not be monitored. The greater significance being, the unobservable code can be tuned to make full use of the hardware resources available at the compiler level. An example of such an observable event could be [8]:

```
S1: every 30 ms
{
S2: receive(Sensor &Sc2, int &Data);
S3: state = newState(State state, int Data);
S4: cmd = nextCmd(State state, int Data);
S5: send(Actuator A1, cmd);
}
```

The observable events that are triggered here are Send and Receive which is constrained to repeat periodically. The other events are not constrained by timing requirements and are dependent only upon CPU and data availability. These event based semantics provide the compiler with enough markers for accurate allocation and scheduling of events based on the timing constraints. For example; the event S3 can be decomposed by applying a transformation and yet another transformation can relocate as much code as necessary to handle a single-period overload. With the timing boundaries established, the compiler has the flexibility to move the code for optimum scheduling.

Once the event-based semantics have been specified at the source code level the compiler’s task now is to transform the code to a low level task set, that is feasible. This task is done is three steps. First, the compiler decomposes the time constrained code into sections based on the control flow. Next, the timing constraints are derived and verified for execution times for these sections. These timing constraints are made available with the help of timing analysis tools. Lastly, code-scheduling transformations are applied to reduce the worst-case sections. This is achieved by moving code from sections with tight timing constraints to sections with more lenient constraints [8].

6. Implementations for Real-Time Operating Systems

Distributed real-time systems are becoming more intertwined with traditional systems that already exist as more and more real-time applications become available. For scheduling such systems, the cyclic executive model which uses a time line analysis is not suitable [12]. It is challenging to test and tune such an executive in a distributed environment. This is further exacerbated by network communication delays.

The new challenge is to develop a kernel for real-time systems which can provide users with a reliable distributed real-time computing environment. One such real-time operating system will be studied to understand how such non-determinism is managed.

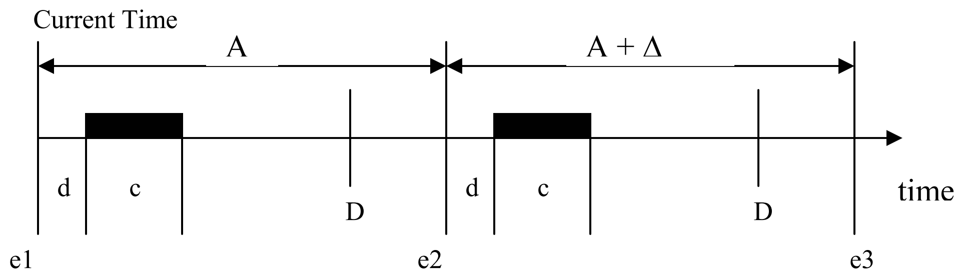
6.1 Real-Time Mach

Carneggie Mellon University's ART (Advanced Real-Time Technology)[12] group has been developing a real-time version of the Mach and the corresponding toolsets for design and analysis. The RT-Mach kernel includes real-time thread management, an integrated time-driven scheduler (ITDS), synchronization, and memory resident objects.

6.1.1 RT-Thread Model

The objective of the thread model is to give full support to the real-time scheduler and provide a uniform system interface to both real-time and non-real-time threads. The ITDS of Mach is based on the RM scheduling paradigm [2, 7].

A real-time thread differs from a non- real-time one, in that, the real-time thread has timing attributes, defined by a timing descriptor. The real-time thread is also defined as a hard real-time and soft real-time thread based on the criticality of the results. The threads can be periodic or aperiodic in nature. Aperiodic threads are defined by the worst case execution time C_j. Some of the timing attributes are shown below:



A: Worst case interarrival time	C: Worst case execution time
D: Deadline	e_i: External event
d: Scheduling delay	

Fig 1. Timing attributes of an aperiodic thread

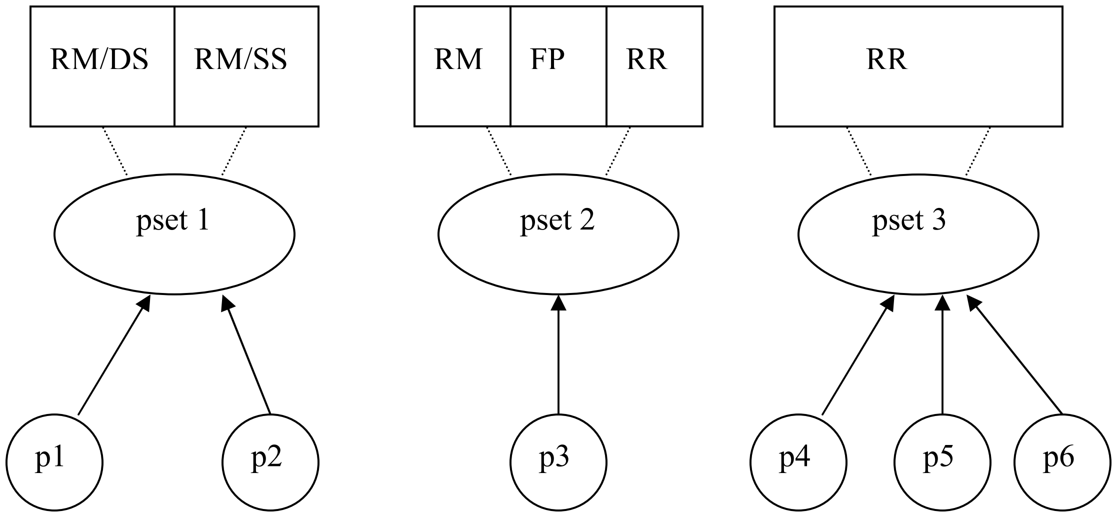
6.1.2 RT-Thread Scheduling

Mach provides a flexible processor allocation facility. This facility uses two kinds of objects: *processor* and *processor set*[12]. A processor object represents a physical processor and the processor set corresponds to a set of processors. A thread belongs to a processor set and a processor belongs to a processor set. A special processor set called the *default processor set* exists. All new processors belong to the default processor set before assignment.

The data structure to manage the scheduling of these threads is called the run queue. All processor set have their own run queues. When a thread becomes runnable, it is added to the run queue for that processor set. Threads do not migrate between disparate processor sets and so different scheduling methods can be chosen for each processor set.

6.1.3 ITDS Scheduling in RT-mach

The integrated time-driven scheduler manages both hard and soft real-time activities. The ITDS scheduler can determine whether a task set can meet its deadline or not. The ITDS adopts a capacity preservation scheme to deal with hard and soft real-time activities. The CPU bandwidth is first given to the hard real-time tasks and the remaining schedulable time is shared between the soft real-time tasks.



RM: Rate Monotonic
FP : Fixed Priority
RR : Round Robin
RM/DS : RM with Deferrable Server
RM/SS : RM with Sporadic Server

P_i : Processor i
P_{set} i : Processor Set i

Fig. 2: ITDS for RT-Mach

Each processor set can have its own scheduling policy as each processor set maintains and manages its own run queue. Therefore, the system can be configured for various real-time applications. For example consider three processor sets: $P_{set}1$, P_{set} 2, P_{set} 3 and six processors P_i.

$P_{set}1$ has two processors: p_1 and p_2, $P_{set}2$ has one processor: p_3, and $P_{set}3$ has three processors: p_4, p_5 and p_6. Now, $P_{set}1$ can execute real-time applications based on RM with deferrable server or sporadic server. $P_{set}2$ executes real-time and non-real-time applications, by changing scheduling policies from RM, fixed priority, or round-robin. $P_{set}3$ only executes non-real-time applications using three processors with round-robin scheduling policy.

7. Discussion

The rate monotonic algorithm is limited to periodic tasks. The drawback of the rate monotonic algorithm is that it assumes tasks are independent, periodic, and preemptible on a single processor system. However, often activities in a real-time system must synchronize and communicate with each other. The rate monotonic scheduling [2, 7] analysis techniques cannot predict the performance of a system with such tasks. The problem of more than one processor connected by a communication medium is a topic of current research. A special case of the synchronization problem described above is the mutual exclusion problem. It is sometimes necessary to enforce mutual exclusion requirements, and usually this means that a task in a critical region cannot be preempted.

The more important challenge in the field of real-time systems is dynamic systems. The stage at which we can predict and guarantee the task completion times. In dynamic systems, both requirements and resource availability keep changing and estimation based on either of these two factors is a daunting task. The issue here is to determine the timing correctness of the scheduler given the resource availability in an unpredictable environment. Handling incomplete computation and fault tolerance are some of the issues here that make dynamic scheduling a real challenge.

Some other issues include the problem of resource management. In real-time systems that include applications, the responsibility of managing the task rests with both the kernel, as well as, the application. In dynamic real-time systems, the task of handling task overloads and fault handling may be with the application. Problems arise, when the designer has to decide how the responsibility is shared between the application and the operating system. One solution to this problem is that the real-time system designer assumes that scheduling is to be handled exclusively by the kernel even when faults occur. Such an approach may not handle certain fault situations leading to failure.

There are many solutions proposed to these problems, however, several challenges remain especially in the area of imprecise computations and algorithm selection before a dynamic real-time system is a viable technology.

8. Conclusion

We have explored the nature of real-time systems in the context of scheduling and its implications on the quality of computation and the behavior of the system. We discussed traditional methods employed to address the problems of scheduling and some practical methods to deal with some of the short comings of assumptions made in such traditional approaches.

In real-time systems scheduling is an integral part of the system operations and most work done in this field ignore the overheads involved in scheduling of tasks. This is a whole new area where there are many challenges and these are to be addressed by researchers.

9. References

[1] J. Xu and D. L. Parnas. "*On satisfying timing constraints in hard-real-time systems.*" IEEE Transactions on Software Engineering, 19(1):70-- 84, January 1993.

[2] C. L. Liu and James W. Layland *"Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment",*1973*,* Project MAC, Massachusetts Institute of Technology and Jet Propulsion Laboratory, California Institute of Technology; Journal of the ACM, 20(1):46--61, 1973

[3] Z. Deng and J. W.-S. Liu, "*Scheduling Real-Time Applications in an Open Environment*," in Proceedings of the 18th IEEE Real-Time Systems Symposium, IEEE Computer Society Press, Dec. 1997

[4] Clifford W. Mercer and Hideyuki Tokuda *"Preemptibility in Real-Time Operating Systems"* School of Computer Science, Carnegie Mellon University

[5] K. Ramamritham and J. A. Stankovic, "*Scheduling algorithms and operating systems support for real-time systems,*" Proceedings of the IEEE, vol. 82, no. 1, pp. 55--67, January 1994.

[6] Jeffay, K., Stanat, D. F., and Martel, C. U. 1991. *"On non-preemptive scheduling of periodic and sporadic tasks."* In Proceedings of the 12 th IEEE Symposium on Real-Time Systems (December 1991), pp. 129--139. IEEE.

[7] J. Lehoczky, L. Sha, and Y. Ding, "*The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior,*" IEEE Real Time Systems symposium, 1989.

[8] Richard Gerber and SeongsooHong, "*Compiler Support for Real-Time Programs*", Department of Computer Science University of Maryland College Park, 1994.

[9] D.B. Stewart and P. Khosla, "*Real-Time Scheduling of Dynamically Reconfigurable Systems*", IEEE International Conference on Systems Engineering, August, 1991, pp. 139-142.

[10] L. Sha and J. B. Goodenough. "*Real-time Scheduling Theory and Ada*." IEEE Computer, pages 53--66, April 1990.

[11] C.W. Mercer and H. Tokuda. "*Preemptibility in real-time operating systems*." In IEEE Real Time Systems Symposuim, pages 78--87, 1992.

[12] H. Tokuda, T. Nakajima, and P. Rao, "*Real-Time Mach: Towards a Predictable Real-Time System*", In Proceedings of USENIX Mach Workshop, October, 1990.

[13] Deadline Scheduling (Embedded Systems Programming) available at http://www.embedded.com/story/OEG20010304S0004

[14] *Real-Time Systems* available at http://www.cse.unl.edu/~goddard/Courses/RealTimeSystems/Lectures/Lectures.html

[15] R. Rajkumar. "*Task Synchronization in Real-Time Systems.*" PhD thesis, Carnegie Mellon University, August 1989.

[16] C. W. Mercer S. Savage H. Tokuda, "`*Applying Hard Real-Time Technology to Multimedia Systems*", Proceedings of the Workshop on the Role of Real-time in Multimedia/Interactive Computing Systems, Raleigh-Durham, NC, December 1993.

[17] Sha, L., Lehoczky, J. P., and Rajkumar, R. "*Solutions for Some Practical Problems in Prioritized Preemptive Scheduling*." In Proceedings of the IEEE Real-Time Systems Symposium. 1986.