A Bandwidth Reservation Algorithm for Multi-Application Systems

Giuseppe Lipari, Giorgio Buttazzo, Luca Abeni
Scuola Superiore S. Anna, Pisa
lipari@sssup2.sssup.it, giorgio@sssup.it, luca@hartik.sssup.it

Abstract

This paper focuses on the problem of providing efficient run-time support for different applications running on a real-time system. The goal of our research is to develop a scheduling algorithm to allow independent applications to coexist on the same system. In order to meet the requested QoS, we assign each application a minimum bandwidth and guarantee that it never demands more. This algorithm does not require the exact knowledge of the execution times and interarrival times of the application tasks: hence, it is especially suited for soft real-time and multimedia applications.

1. Introduction

Most of today's computer systems (including conventional PCs) are used to run concurrent applications, some of them consisting of several concurrent tasks. Some application may be subject to timing constraints, which can be hard (as in control systems) or soft (as in multimedia systems). To guarantee hard timing constraints and provide a predefined Quality of Service (QoS) to soft real-time applications, the operating systems must support suitable resource reservation mechanisms and scheduling algorithms.

Traditional operating systems are not likely to meet this goal as they do not manage system resources according to the requirements of soft real-time streams, because scheduling decisions are not based on parameters like the *deadlines* or the *rates* of the application tasks. On the other hand, the use of a hard real-time system for handling soft real-time applications can be inappropriate, because it may be very difficult to provide a precise estimation of worst case execution times (WCETs) for multimedia tasks, even if the application always runs on the same hardware. Moreover, in a hard real-time environment, if the WCET of a task is not correctly estimated, it could cause the failure of all system tasks. Rather, in a multi-application system, it would be desirable that only the failed application is affected.

In [9], Mercer, Savage, and Tokuda propose a scheme based on CPU capacity reserves, where a fraction of the CPU bandwidth is reserved to each task. This solution is not suitable for our purposes, since they assign a bandwidth to each single task, and not to the entire application. Moreover, if a task instance executes for more than its WCET, the remaining portion of the instance is scheduled in background, prolonging its completion of an unpredictable time.

In [5], Kaneko et al. propose a scheme based on a periodic process (the multimedia server) dedicated to the service of all multimedia requests. This allows to nicely integrate multimedia tasks together with hard real time tasks; however, being the server only one, it is not easy to control the QoS of each application.

In [3], Liu and Deng describe a scheduling hierarchy which allows hard real-time, soft real-time and non real-time applications to coexist in the same system, and to be created dynamically. According to this approach, which uses the EDF scheduling algorithm as a low-level scheduler, each application is handled by a dedicated server. This solution can be used to isolate the effects of overloads at the application level, but requires the knowledge of the WCET even for soft and non real-time tasks.

Recently, some scheduling algorithms have been proposed [10, 4] to mix some form of real-time support with a notion of fairness. In these works, the isolation is realized allocating the processor to applications in a weighted fair manner. These methods are effective in scheduling soft real-time tasks, and nicely integrates with the existing software: for example, in [4] the authors realized a two-level scheduling algorithm and it is possibly to select the higher-level scheduling policy that best meets the application requirements. However, this approach may be unsuitable for guaranteeing applications consisting of hard real-time tasks, because the method is quite sensitive to the temporal quantum seen by an application, which depends on the percentage of processor reserved to it.

In this paper, we propose a scheduling methodology based on reserving a fraction of the processor bandwidth to each application (in a way similar to processor capacity reserves of Mercer [9]). However, to efficiently handle the problem of task overloads, each application is scheduled by a dedicated server, which assigns suitable deadlines to the

served tasks whenever the reserved time is consumed. The proposed method does not require the exact knowledge of the WCETs.

The paper is organized as follows: Section 2 introduces the terminology and the assumptions used throughout the paper; Section 3 describes the main goals of this work; Section 4 formally presents the algorithm; Section 5 illustrates the properties of the algorithm; and Section 6 presents our conclusions.

2. Terminology and assumptions

A task τ_i is a stream of requests for a shared resource (the CPU), each of them characterized by a request time $r_{i,j}$ and an execution time $c_{i,j}$:

$$\tau_i = \{(r_{i,0}, c_{i,0}), (r_{i,1}, c_{i,1}), \ldots\}.$$

Statistically, a task can be described by a couple (c_i, p_i), where c_i and p_i are random variable with probability distribution functions (PDF) $P\{r_{i,j+1} - r_{i,j} = p_i\}$ and $P\{c_{i,j} = c_i\}$, respectively.

This characterization is very general and includes most of the task models presented in the literature. For example, in a periodic task model, $\forall j \geq 0, \quad r_{i,j+1} = r_{i,j} + P_i$, where P_i is a fixed parameter; in a sporadic task model, $\forall j \geq 0, \quad r_{i,j+1} \geq r_{i,j} + MIT_i$, where MIT_i is the *minimum interarrival time*. Usually, a schedulability test is performed based on worst-case analysis: given the *worst case execution times* ($\text{WCET}_i = max_j(c_{i,j})$), and the MITs, it is possible to test if each task will meet its deadline.

In our model, we do not make any assumption on the arrival times and on the execution times of each instance, nor on the WCETs and the MITs. Each task is assigned just one constant parameter T_i which represents the "desired" interarrival time between two consecutive instances.

The system designer can assign this parameter based on the temporal behavior of the task. For example, if for each task the WCET is known, and T_i is assigned a value equal to the MIT, then our model is equivalent to the Liu and Layland model and a schedulability test can be easily performed. On the contrary, if the system designer does not have a precise knowledge of the task, T_i can be set based on mean values.

An application is a set $\mathcal{A}_j = \{\tau_i | i = 1, \ldots n_j\}$ of cooperating tasks. Tasks belonging to the same application are allowed to share resources, while tasks from different applications are not. Our system consists of a set of applications running on a single processor.

A *server* is an algorithm which, given a task request, splits it in one or more *jobs*, and assigns each of them a scheduling deadline. Our server mechanism will be described in Section 4.

The formal properties of the server will be analyzed using the *processor demand approach* ([1] pag. 102). Let us define $D_i(t_1, t_2)$ as the amount of execution time demanded by task τ_i in the interval $[t_1, t_2]$:

$$D_i(t_1, t_2) = \sum_{t_1 \leq r_{i,j}, t_2 \geq d_{i,j}} C_{i,j},$$

where $r_{i,j}$, $d_{i,j}$, $C_{i,j}$ are the request time, the assigned deadline, and the execution time of the j-th job of τ_i, respectively.

The *Demand Bound Function* $dbf(\mathcal{A}, t)$ is the maximum amount of time that all the tasks $\tau_i \in \mathcal{A}$ demand in any interval of length L:

$$dbf(\mathcal{A}, L) = \max_{t \geq 0} \Big(\sum_{\tau_i \in \mathcal{A}} D_i(t, t + L) \Big).$$

Finally, let us define U^A as the *bandwidth* of set $\mathcal{A}$:

$$U^A = \max_{L > 0} \left(\frac{dbf(\mathcal{A}, L)}{L} \right).$$

3. Goals of this research

The main goal of this paper is to present a two-level scheduler to handle multi-task applications. At the lowest level we use the EDF algorithm to schedule the tasks of the system applications. Hence, a scheduling deadline needs to be assigned to each task. This is done by a server which implements the high-level scheduler: the server receives the task requests, splits them into jobs and assigns a deadline to each job. The server mechanism is built to meet the following goals:

- **Bandwidth isolation**: each application $\mathcal{A}$, served by a server S, is assigned a bandwidth U^A and it is guaranteed that it will never demand $U_a > U^A$. All tasks in $\mathcal{A}$ share the same bandwidth U^A.
- **Hard schedulability**: Given an application $\mathcal{A}$, served by a server S with bandwidth U^A, if all the parameters are correctly estimated (i.e. $\forall \tau_i \in \mathcal{A}$ the WCET_i is known and $T_i \leq \text{MIT}_i$) then the application is feasible (i.e., each task will complete before its relative deadline) if and only if:

$$\sum_{\tau \in \mathcal{A}} \frac{\text{WCET}_i}{T_i} \leq U^A. \qquad (1)$$

The *bandwidth isolation* property ensures that an application cannot affect the schedulability of the others. In fact, even if the task parameters are not correctly estimated, the server mechanism must impose that the demanded bandwidth does not exceed U^A, so a bandwidth $1 - U^A$ is left to

the other applications: in this case, only the "wrong" application slows down.

On the other hand, the *hard schedulability* property ensures that, given an application $\mathcal{A}$, if the task parameters are correctly estimated, an a-priori guarantee can be performed: the application executes as it is running alone on a slower processor. Notice that the schedulability condition given by equation (1) is equivalent to the Liu and Layland condition [8], where U_{lub} is substituted with U^A.

In this framework, the QoS of an application depends on the precision of the estimates T_i and on the bandwidth U^A assigned to application. If the system designer performs a precise analysis and calculates WCET$_i$ and MIT$_i$ the application can be guaranteed as a hard real-time application; on the other hand, if the temporal constraints are not critical (like in multimedia tasks), the bandwidth can be allocated based on mean values, in order to obtain a better CPU utilization.

It is worth to note that, for our purposes, it is not important to obtain an isolation between tasks of an application $\mathcal{A}$, but only between tasks belonging to different applications. Thus, a task $\tau_i \in \mathcal{A}$ which requires too much computation time can affect only the other tasks in $\mathcal{A}$.

4. Description of the server

In order to bound its bandwidth U^A, each application $\mathcal{A}$ is handled by a dedicated server which keeps track of the time already consumed by the application tasks: whenever a task $\tau_i \in \mathcal{A}$ executes, it gives a contribution to the consumed bandwidth. The server dynamically modifies the tasks' deadlines such that the total processor demand never exceeds U^A.

The server uses the following variables:

- $d_i(t)$ is the deadline for the current instance of task τ_i;
- $r_i(t)$ is the release for the current instance of task τ_i;
- $C(t)$ is the server capacity at time t ($C(0) = 0$);
- $E(t)$ is the execution time of the current executing task, measured from the last time it was scheduled ($E(0) = 0$);
- U is the bandwidth assigned to the application;
- $L_d = \{(d_0, R_0), \ldots, (d_m, R_m)\}$ is an ordered list of pairs (d, R), where d represents a deadline and R the maximum time that application tasks can execute with deadline less than or equal to d. The list is ordered by increasing d. The list is consistent if the maximum execution time R_k permitted within each deadline d_k is not decreasing with d_k, that is:

$$\forall (d_j, R_j), (d_k, R_k) \in L_D \mid d_j \leq d_k \Rightarrow R_j \leq R_k.$$

Two operations are defined on L_d: $insert(L_d, (d, R))$, which inserts the pair (d, R) in L_d in the right order, and $extract(L_d, (d, R))$, which extracts the pair (d, R) from L_d. At time $t = 0$, the list is empty.

These variables are private to each server (and, hence, to each application). So, we should write $C^A(t)$ to indicate the capacity of the server dedicated to application $\mathcal{A}$. For the sake of clarity, from now on we will describe the behavior of a single server at a time; thus, we simply use $C(t)$ to denote the capacity of the current application server.

The rules used by the server to share a bandwidth U among among the application tasks are listed below.

1. If task τ_i arrives at t, then
$$d_i(t) = t + T_i.$$
2. If at time t task τ_i is scheduled to execute, then server assigns it a maximum execution time $C(t)$ according to the specified bandwidth U and the current L_d status. In particular:
   (a) If L_d is empty:
   $C(t) = T_i U;$
   $insert(L_d, (d_i(t), C(t)));$
   (b) If $\exists (d_k, R_k)$ such that $d_i(t) = d_k$, then
   $C(t) = R_k = min(R_k, T_i U);$
   $\forall (d_j, R_j) \mid d_j < d_i(t) \;\; R_j = min(R_j, R_{j+1});$
   (c) if $d_k < d_i(t) < d_{k+1}$ then
   $C(t) = min(R_k + (d_i(t) - d_k)U,\ T_i U,\ R_{k+1});$
   $insert(L_d, (d_i, C(t)));$
   $\forall (d_j, R_j) \mid d_j < d_i(t) \;\; R_j = min(R_j, R_{j+1});$
   In any case,
   $\forall (d_j, R_j) \in L_d \mid d_j \leq r_i \;\Rightarrow\; extract(L_d, (d_j, R_j)).$
3. If τ_i is the task with the earliest deadline, τ_i can execute for at most $C(t)$ before it is preempted or it completes. Suppose that τ_i executes for Δt, then:
$$\begin{aligned} C(t + \Delta t) &= C(t) - \Delta t; \\ E(t + \Delta t) &= E(t) + \Delta t. \end{aligned}$$
4. If at time t task τ_i leaves the processor (because it is preempted or it has finished), then list L_d is updated as follows:
   $\forall (d_k, R_k) \in L_d \mid d_k \geq d_i(t) \;\; R_k = R_k - E(t);$
   $E(t) = 0;$
   $\forall (d_j, R_j) \in L_d \mid d_j < d_i(t) \;\; R_j = min(R_j, R_{j+1}).$

5. If $C(t) = 0$ and τ_i is the executing task, then the bandwidth assigned to that task is exhausted; hence its deadline must be postponed:
   - τ_i is extracted from the EDF queue and Rule 4 is applied;
   - $d_i(t) = d_i(t) + T_i$;
   - τ_i is inserted in the EDF queue; if it is still the earliest deadline task, Rule 2 is applied and τ_i scheduled.

Now, let's get a closer look to the algorithm. At time $t = 00$, L_d is empty. When a task τ_i arrives, Rule 1 assigns it a deadline and puts it in the EDF queue. The task may not execute immediately because other tasks from other applications could be active in the system. When τ_i becomes the earliest deadline task, Rule 2 is applied, and a capacity equal to the task period multiplied by the server bandwidth is assigned. Then, a pair $(d_i(t), C(t))$ is inserted in L_d. In this way, the total execution of all the tasks of the same application with deadline less than or equal to $d_i(t)$ cannot be more than $C(t)$. Every time a task of the application is executed, the capacity is decremented (Rule 3). Suppose that, at time $t_1 > t$, task τ_j becomes the earliest deadline task and $d_j(t_1) < d_i(t)$. Then a preemption occurs, τ_i is descheduled, Rule 4 is applied, and list L_d is updated. When τ_j is scheduled, Rule 2 is applied; in this case the capacity cannot be more than the residual capacity for $d_i(t_1)$.

An example of schedule produced by the algorithm on two application is shown in Figure 1. The first application, $\mathcal{A}$, contains one task, $\tau_1^A = \{(5, 5)\}$ (i.e., a single job arriving at time $r_1^A = 5$ and requesting $c_1^A = 5$ units of time). Assume that a bandwidth $U^A = 0.5$ is assigned to $\mathcal{A}$ and a period $T_1^A = 10$ is assigned to τ_1^A. The second application, $\mathcal{B}$, contains two tasks, $\tau_2^B = \{(3, 4), (11, 2)\}$ and $\tau_3^B = \{(0, 4)\}$; assume that a bandwidth $U^B = 0.5$ is assigned to $\mathcal{B}$, and periods $T_2^B = 8$ and $T_3^B = 20$ are assigned to the tasks. Let's focus our attention on application $\mathcal{B}$.

At time $t = 0$, task τ_3^B starts executing; Rules 1 and 4 are applied: $C^B(0) = T_3^B \cdot U^B = 10$. At this point, $L_d^B = \{(20, 10)\}$. At time $t = 3$, task τ_3^B is preempted by τ_2^B: Rule 4 is applied and list L_d^B is updated as $L_d^B = \{(20, 7)\}$. When τ_2^B is scheduled, Rule 2 is applied, the capacity of the $\mathcal{B}$ server is computed as $C^B(3) = min\{T_2^B * U^B, 7\} = 4$, and a new element is added to the list: $L_d^B = \{(11, 4), (20, 7)\}$. At time $t = 7$, τ_2^B finishes and Rule 4 is applied again, so the $\mathcal{B}$ list becomes $L_d^B = \{(11, 0), (20, 3)\}$. What the list indicates is that, at time $t = 7$, no task from application $\mathcal{B}$ can execute with deadline less than or equal to 11, whereas they can execute for a maximum of 3 time units with a deadline in the interval $(11, 20]$.

At time $t = 7$, task τ_1^A starts executing, but it belongs to application A, so nothing changes for server $\mathcal{B}$. At time $t = 11$, the second request of τ_2^B arrives and it is assigned a deadline equal to $d_2^B(7) = 19$ (Rule 1). At time $t = 12$, τ_2^B starts executing, hence $L_d^B = \{(11, 0), (19, 3), (20, 3)\}$ and $C(12) = 3$ (Rule 2). At time $t = 14$, τ_2^b is descheduled, so Rule 4 is applied and $L_d^B = \{(11, 0), (19, 0), (20, 1)\}$. Now, τ_3^B is resumed and it can execute for at most 1 unit of time: L_d^B does not change and $C^B(t) = 1$. Finally, at time $t = 15$, τ_2^B finishes when the capacity is completely exhausted.

Another example is shown in Figure 2. Here, there is only a single application $\mathcal{A}$ consisting of 3 tasks: $\tau_1 = \{(5, 5)\}$ which is assigned $T_1 = 4$; $\tau_2 = \{(2, 6)\}$ which is assigned $T_2 = 8$ and $\tau_3 = \{(0, 4)\}$ which is assigned $T_3 = 20$. A bandwidth $U^A = 0.75$ is reserved to application $\mathcal{A}$. Notice that at $t = 5$, $L_d^A = \{(9, 3), (10, 3), (20, 10)\}$, hence τ_1^A (having a deadline equal to 9) is allowed to execute for at most $C^A(t) = 3$ time units. At time $t = 8$ the task has not finished yet, but the capacity is exhausted. In this case, Rule 5 is applied, the deadline of τ_1^A is postponed at time $t = 13$ and $L_d^A = \{(9, 0), (10, 0), (20, 7)\}$. At this point, τ_2^A is resumed, but the capacity is $C^A(8) = 0$! Again, Rule 5 is applied and the deadline of τ_2^A is postponed at time $t = 18$. Finally, τ_1^A is scheduled, list L_d is updated at $\{(9, 0), (10, 0), (13, 9/4), (20, 7)\}$ and $C^A(8) = 9/4$.

5. Formal Properties

The proposed algorithm verifies both the *bandwidth isolation* and the *hard schedulability* properties, introduced in Section 3. More formally, the following results can be proven (the formal proofs of these theorems are not reported here and can be found in [7]).

Theorem 1 *If application $\mathcal{A}$ is assigned a bandwidth U^A, our server guarantees that, in any time interval of length L, the application tasks will never demand more than $L \cdot U^A$:*

$$\forall L > 0 \;\; dbf(\mathcal{A}, L) \leq L \cdot U^A$$

Theorem 2 *An application $\mathcal{A}$ with bandwidth U^A, served by a server S, is feasible (i.e., each task will complete before its relative deadline) if and only if:*

$$\sum_{\tau \in \mathcal{A}} \frac{WCET_i}{T_i} \leq U^A.$$

The *Bandwidth Isolation Property* stated in Theorem 1 allows to mix different applications in the same computer system and ensures that the temporal behavior of an application is not affected by, nor interferes with the others. Moreover, this property is very useful in highly dynamic systems, where is difficult to obtain a priori information about applications. Even if the request times and the execution times

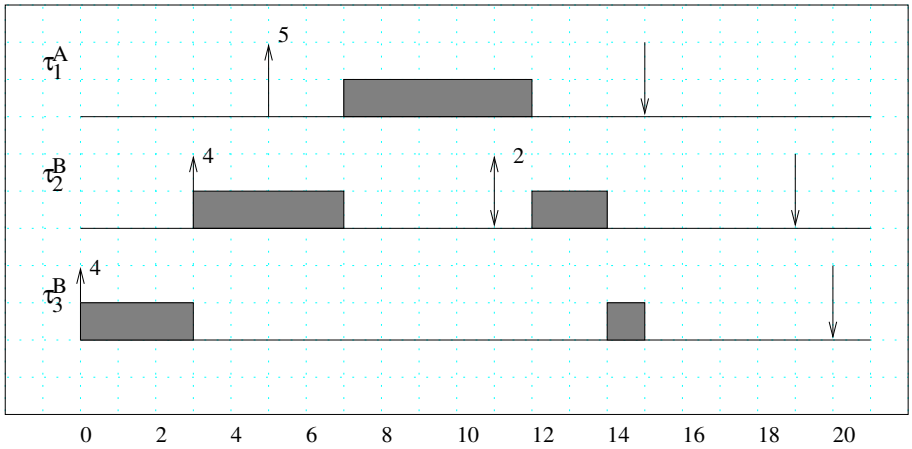

Figure 1. Two tasks with preemption.

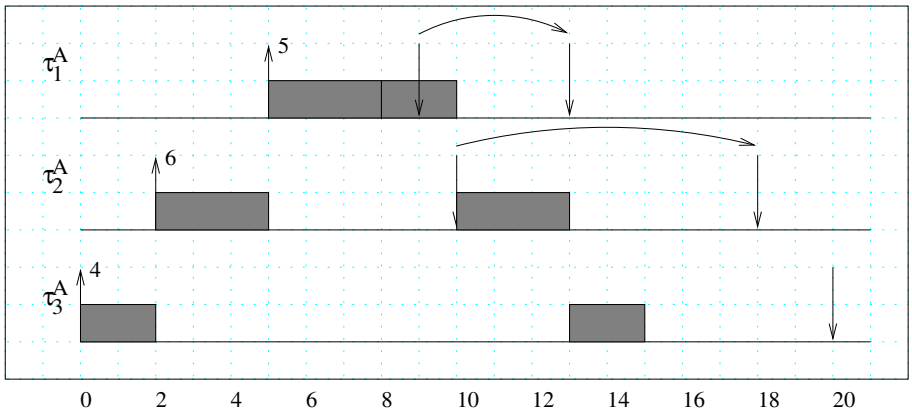

Figure 2. Two nested preemptions.

of the tasks of an application are not known in advance, it is guaranteed that the other applications are not affected.

The *Hard Schedulability Property* stated in Theorem 2 allows to guarantee hard real-time tasks within a single application, even though other soft real-time applications run concurrently in the same system.

In addition, an arbitrary QoS scheme can easily be implemented on top of our algorithm. For example, an application such as a MPEG player could request a QoS Manager a desired QoS level by specifying its bandwidth and the rates of its tasks with some heuristic rule. If there is enough free bandwidth in the system, the application is admitted, otherwise it is rejected. At run-time, an application could monitor the QoS parameters, and possibly re-negotiate its bandwidth if they are not satisfactory, or decrease the tasks' rates, depending on the scheme adopted.

6. Conclusions

In this work, we presented a mechanism for scheduling real-time multi-task applications on a uniprocessor system. The main goal of this research is to reserve and guarantee a bandwidth to each application, independently from the execution requests of the tasks. This is achieved by managing each application through a dedicated server which assigns suitable deadlines to application tasks' and put them in a single EDF queue. If all the parameters are correctly estimated, our service method guarantees that the resulting schedule is equivalent to the one obtained by EDF. On the contrary, if the parameters of some task are not correctly estimated, only the tasks belonging to the application which the failed task belongs to will be affected.

In the next future, we plan to implement the proposed mechanism in the HARTIK kernel [2, 6] and use it as a basic support for multimedia applications.

References

[1] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Boston, 1997.

[2] G. C. Buttazzo. Hartik: A real-time kernel for robotics applications. In *IEEE Real-Time Systems Symposium*, December 1993.

[3] Z. Deng and J. W. S. Liu. Scheduling real-time applications in open envirovment. In *IEEE Real-Time Systems Symposium*, San Francisco, December 1997.

[4] P. Goyal, X. Guo, and H. M. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *2nd OSDI Symposium*, October 1996.

[5] H. Kaneko, J. Stankovic, S. Sen, and K. Ramamritham. Integrated scheduling of multimedia and hard real-time tasks. In *Proceedings of Real-Time Systems Symposium*, December 1996.

[6] G. Lamastra, G. Lipari, G. Buttazzo, A. Casile, and F. Conticelli. Hartik 3.0: A portable system for developing real-time applications. In *Real-Time Computing Systems and Applications*, October 1997.

[7] G. Lipari. An algorithm for resource reservation in multi-applications systems. Technical Report RETIS TR-98-01, Scuola Superiore S. Anna, 1998.

[8] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1), 1973.

[9] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves for multimedia operating systems. Technical Report CMU-CS-93-157, Carnegie Mellon University, Pittsburg, May 1993.

[10] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *IEEE Real Time System Symposium*, 1996.