A Bandwidth-Based Server for Multimedia Applications

Luca Abeni, Giuseppe Lipari, Giorgio Buttazzo
Scuola Superiore S. Anna, Pisa
luca@hartik.sssup.it, {lipari, giorgio}@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 two-level 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: for this reason, it is especially suited for multimedia applications.

1. Introduction

One of today's most exciting research field is providing support for multi-task soft real-time applications (like multimedia applications) running on a conventional PC together with traditional ones. For this kind of applications, it is important to provide a predefined Quality of Service (QoS); however, traditional operating systems are not likely to meet this goal as they do not manage system resources according to the requirements of multimedia streams. In fact, 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 task, 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.

Recently, some scheduling algorithms have been proposed [1, 2] to mix some form of real-time support with a notion of fairness, but they do not make use of conventional real-time theory.

In [3], 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.

In [4], 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.

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 [3]). 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.

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 behaviour 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 $\mathcal{A}_j = \{\tau_i | i = 1, \dots n_j\}$ 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.

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:

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

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_s and it is guaranteed that it will never demand $U_a > U_s$. All tasks in $\mathcal{A}$ share the same bandwidth U_s.

- **Hard schedulability**: Given an application $\mathcal{A}$, served by a server S with bandwidth U_s, 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) iff:
$$\sum_{\tau \in \mathcal{A}} \frac{\text{WCET}_i}{T_i} \leq U_s. \quad (1)$$

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_s, so a bandwidth $1 - U_s$ 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 [5], where U_{lub} is substituted with U_s.

In this framework, the QoS of an application depends on the precision of the estimates T_i and on the bandwidth U_s 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 the bandwidth U_a required by application $\mathcal{A}$, a server 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_s.

More formally, at each time t the server state can be described by the following variables:

- $u(t)$ is the *residual bandwidth* of the server; $0 \leq u(t) \leq U_s$. It represents the actual amount of bandwidth not yet consumed by the application tasks, and it is decreased by $1/T_i$ for each unit of time task τ_i executes.

- $d_i(t)$ is the last deadline assigned by the server to task τ_i.

- $\rho_i(t)$ is the *job density*. It represents the fraction of computation time, with respect to T_i, consumed by the current active instance of τ_i. That is, indicating with $e_i(t)$ the amount of time executed by the current active instance up to t, $\rho_i(t) = \frac{e_i(t)}{T_i}$, with $0 \leq \rho_i(t) \leq U_s$.

- $\eta_i(t)$ is the *task recharge*. It represents the bandwidth consumed by τ_i before the last release time. When the current active instance of τ_i finishes, the algorithm set $\eta_i(t) = \eta_i(t) + \rho_i(t)$ and $\rho_i(t) = 0$. This variable is used to recharge the total bandwidth of the server: when $u(t) = 0$, $u(t) = u(t) + \eta_i(t)$.

At the beginning, $u(0) = U_s$; $d_i(0) = 0$; $\rho_i(0) = 0$; $\eta_i(0) = 0$.

The key idea behind the algorithm is to avoid that the whole application can execute for more than $L \cdot U_s$ in any interval of length L, independently on the single application task execution time (in some sense, tasks belonging to the same application share the same bandwidth). The algorithm does not make any assumption on the actual task request: when a task consumes all the bandwidth ($u(t)$ reaches 0), the server tries to recharge it, and if it is not possible (i.e., if $\eta_i = 0$), the task deadline is postponed at $d_i = d_i + T_i$.

An example of schedule obtained using this method, is shown in Figure 1. The system consists of two applications: $\mathcal{A}$ and $\mathcal{B}$. $\mathcal{A}$ contains only task τ_{a1} which is a periodic task: it is characterized by a constant interarrival time $P_{a1} = 6$ and by a constant computation time $C_{a1} = 2$. It is assigned parameter $T_{a1} = 6$, and a total bandwidth of $U_{s1} = C_{a1}/T_{a1} = 1/3$. The second application $\mathcal{B}$ contains two tasks, τ_{b1} and τ_{b2}. τ_{b1} is periodic with a constant interarrival time $P_{b1} = 8$, but does not have a constant execution time; it is assigned $T_{b1} = 8$. τ_{b2} neither have a constant interarrival time, or a constant execution time; it is assigned $T_{b2} = 12$; a bandwidth of $U_{s2} = 2/T_{b1} + 5/T_{b2} = 2/3$ is assigned to application $\mathcal{B}$. Since $U_{s1} + U_{s2} = 1$, the system is feasible. Figure 1 shows the resulting EDF schedule, the function $u(t)$, and the values of η_i each time they vary (for clarity, values are multiplied by 24).

Let us focus on application $\mathcal{B}$, and in particular on task τ_{b2}. As at time $t = 0$ task τ_{b2} issues a request of 6 units of computation time, at time $t = 9$ the server bandwidth $u(9)$ is exhausted whereas the task asks to execute for one more units. Since $\eta_{b2} = 0$, the deadline is postponed at $d_{b2} = 24$ and $\eta_{b2} = \rho_{b2} = 10/24$. This is equivalent to creating a new job of τ_{b2} with deadline at 24. However, we cannot recharge immediately the server bandwidth $u(t)$ of $\eta_{b2} = 10/24$, otherwise the task τ_{b1} will receive too much service. This recharge is done only at time $t = 15$, when task τ_{b2} tries to execute and $u(15) = 0$. In this case, the deadline of τ_{b2} remains at 24. It is worth to notice that we do not need to know the WCETs in order to make scheduling decision, but only if we want to guarantee the schedulability of the application with equation (1) (just like in EDF).

The algorithm is formalized by the following rules.

1. if task τ_i executes for Δt units of time, and $u(t) \geq \frac{\Delta t}{T_i}$, then
$$u(t + \Delta t) = u(t) - \frac{\Delta t}{T_i};$$
$$\rho_i(t) = \rho_i(t) + \frac{\Delta t}{T_i};$$

2. when an instance of τ_i terminates, $\eta_i(t) = \eta_i(t) + \rho_i(t); \quad \rho_i(t) = 0;$

3. if $u(t) = 0$ and τ_i is the current active instance,
   - if $\eta_i(t) \neq 0$ then $u(t) = u(t) + \eta_i(t);\ \eta_i(t) = 0;$
   - if $\eta_i(t) = 0$ then $\eta_i(t) = \eta_i(t) + \rho_i(t);\ \ \rho_i(t) = 0;\ \ d_i(t) = d_i(t) + T_i;$

4. if task τ_i issues a request at time t, a new suitable deadline is assigned to it:
$$d_i(t) = \max(d_i(t), t) + T_i.$$

Some things are worth to notice: a task contributes to the bandwidth consumed by the application with a weight $\frac{1}{T_i}$ to be multiplied by the actual execution time (rule 1) (notice that, in Figure 1 the slope of function u(t) depends on which task is executing). This bandwidth is cumulated in ρ_i and, when the instance finishes, it is "stored" in η_i in order to recharge the total bandwidth at the opportune instant (rule 2). In fact, in order not to allow tasks to execute for more than they are permitted, the bandwidth $u(t)$ cannot be recharged at arbitrary times.

For this reason, the recharging rule 3 is a bit complicated. Firstly, the recharge is done only when $u(t) = 0$. Intuitively, if $\eta_i(t) \neq 0$, then the recharge can be done without postponing the deadline. In the other case ($\eta_i(t) = 0$), the task deadline must be postponed; it is just like creating another job of task τ_i. Finally, when a task issues a request, a suitable deadline must be assigned to it (rule 4).

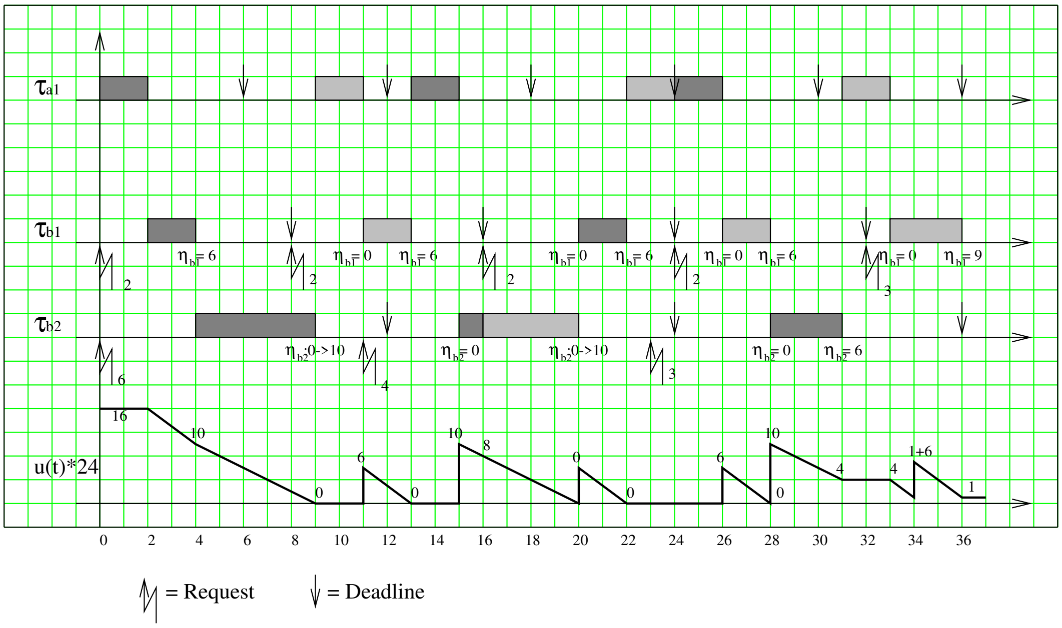

Figure 1. Example of schedule.

As shown in the previous example, tasks belonging to the same application share some server bandwidth: in fact, at time $t = 31$, task τ_{b2} finishes earlier, so task τ_{b1} can execute for the 3 units of time requested without postponing its deadline. In some sense, it "steals" a bit of bandwidth from τ_{b2}. Similarly, if at time $t = 0$ τ_{b1} requests 3 instead of 2, it "steals the bandwidth" to τ_{b2} which in turn can execute less. By our point of view, this is not a pitfall, because we want to let tasks from the same application share the bandwidth: in the example, τ_{b1} can use the bandwidth left by τ_{b2} (which belongs to the same application) for executing a units more, whereas τ_{a1} cannot.

5. Conclusions

In this work, we presented a mechanism for scheduling real-time multitask applications on a uniprocessor system. The main goal is reserving and guaranteeing a bandwidth to each application, independently from the execution requests of the tasks. Each application has a dedicated server which assigns suitable deadlines to application tasks' and put them in an 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 same application of the failed task will be affected.

We described the main properties of the server and proposed a possible set of rules. By now, we are experimenting the performances of our algorithm with simulations. In the next future, we plan to simplify the rules as much as possible and evaluate some alternative. Once the server mechanism will be well-defined and evaluated, it will be implemented in an existing real-time kernel [6, 7] to be used as the base for an operating system designed to support multimedia applications.

References

[1] Ian Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes E. Gehrke, and C. Greg Plaxton, "A proportional share resource allocation algorithm for real-time, time-shared systems," in *IEEE Real Time System Symposium*, 1996.

[2] Pawan Goyal, Xingang Guo, and Harrik M. Vin, "A hierical cpu scheduler for multimedia operating systems," in *2nd OSDI Symposium*, October 1996.

[3] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda, "Processor capacity reserves for multimedia operating systems," Tech. Rep. CMU-CS-93-157, Carnegie Mellon University, Pittsburg, May 1993.

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

[5] C.L. Liu and J.W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the Association for Computing Machinery*, vol. 20, no. 1, 1973.

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

[7] 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.