WP4 – Resource Management Components

Deliverable D4.3
Definition of new RM functionalities

WP4 – Resource Management Components : Deliverable 4.3 – Definition of new RM functionalities
by Giuseppe Lipari

Published July 2003

Table of Contents

Document Presentation

Project Coordinator

Organisation:	UPVLC
Responsible person:	Alfons Crespo
Address:	Camino Vera, 14, 46022 Valencia, Spain
Phone:	+34 963877576
Fax:	+34 963877576
Email:	alfons@disca.upv.es

Participant List

Role	Id.	Participant Name	Acronym	Country
CO	1	Universidad Politecnica de Valencia	UPVLC	E
CR	2	Scuola Superiore Santa Anna	SSSA	I
CR	3	Czech Technical University in Prague	CTU	CZ
CR	4	CEA/DRT/LIST/DTSI	CEA	FR
CR	5	Unicontrols	UC	CZ
CR	6	MNIS	MNIS	FR
CR	7	Visual Tools S.A.	VT	E

Document version

Release	Date	Reason of change
1_0	15/01/2003	First release

1 Introduction

1.1 Resource Reservation scheduling components

Until now, we have developed two schedulers (CBS [Abe98] and GRUB [Lip00]), a resource manager (the feedback scheduler), a user library and monitoring tools. The two schedulers are able to provide temporal isolation and real-time guarantees to soft real-time tasks as well as to legacy Linux processes. In addition, the GRUB scheduler is able to reclaim unused bandwidth. The reclaimed bandwidth can be used for two purpouses:

- to give more bandwidth to processes that need to execute more
- to save energy by reducing the frequency of the processor.

However, both schedulers suffer some problem in certain situation. When using these schedulers to execute a non-periodic legacy Linux application a particular problem that we call *deadline aging* can happen. We will explain this problem in more detail in the next section and propose one possible solution that will be implemented in the next version of the Resource Management components.

1.2 Resource Management component

We would like also to improve the behavior of the resource management component. In the current version, it is a dynamically loadable module in the Linux kernel that, using the features of a resource reservation scheduler, dynamically adjusts the server bandwidth to maximise the quality of service experienced by the application. The algorithm is based on a feedback control strategy: it measures the *scheduling error* (i.e. The difference between the task deadline and the task finishing time) and tries to reduce this error to 0 by incrementing/decrementing the server maximum budget.

The parameter of the feedback controller can be customised to the application characteristics. If the application complies to some requirements (for example, the maximum variation of the computation time is bounded) then our feedback controller guarantees certain properties. See [Pal03] and [Abe02]for a complete description of the controller and its properties.

The current version of the controller is quite generic, in the sense that it does not exploit the characteristic of the application. In fact, it has been designed to work well with a wide range of applications.

It would be nice if we could customise the feedback controller to the characteristic of the application. In the context of the OCERA project, we are going to apply feedback scheduling techniques to multimedia applications like an MPEG player. It is well know that a MPEG stream consist of an almost regular pattern of frames. We would like to exploit this regular pattern to improve the behavior of the feedback scheduler. In section 3 we propose a possible improvement of the feedback scheduler and present some preliminary simulation result that is very encouraging.

2 Problems with CBS and GRUB

To understand the problem, it is necessary to know how algorithms CBS and GRUB schedule the processes. To avoid repetition, we remand to deliverables D4.1 and D4.2. The original formulation of the two algorithms can be found in [Abe98] and in [Lip00], respectively.

The Constant Bandwidth Server does not behave very well when serving a non-periodic process that consist of one single instance. For example, when we execute the compiler, it runs without stopping until the program has been compiled. Many “batch” programs have a similar behaviour. The problem is explained by the following example.

Example 1. Suppose that the two processes are in the system. The first process is a non-periodic process with a single instance. It is activated at time 0, and it is server by a CBS with a budget $Q_1=1$ and a period $P_1=4$. The second process is again a non-periodic process with a single instance, server by another CBS with budget $Q_2=3$ and period $P_2=6$ and it is activated at time $t=7$. The schedule is shown in Figure 2.1.

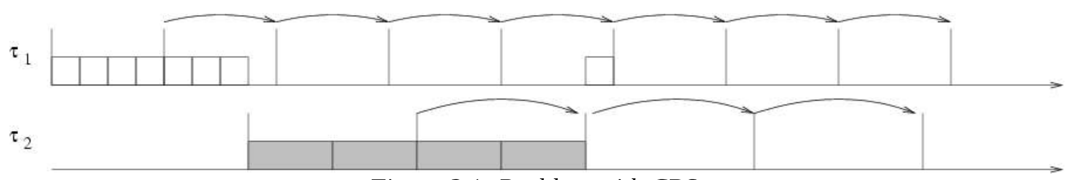

Figure 2.1: Problem with CBS.

As you can see, at the beginning only the first process is active. This process consume all its budget immediately, so its deadline is postponed. However, since it is the only active process, it continues to execute and consume again all its budget, postponing its deadline. Very soon, its deadline goes very far: after 7 instances, the deadline is $d=28$. When the second process arrives, it is the one with the earliest deadline and it executes. When its budget is entirely consumed, its deadline is postponed: however, since the deadline of the first process is very far, it is still the earliest deadline process and continues to execute. It is only after its own deadline becomes greater than the first process deadline that it is preempted and the first process can continue to execute. We say that both processes suffer of *deadline aging*. This problem can happen if at some point the system is not fully loaded. Since there is nothing else to do, the first process continues to execute, but in doing so it consumes its future budgets and its deadline is postponed very often. As a result, when the second process arrives, the priority of the first process is so low that it executes as it were in background.

The previous behavior is not desirable for many reason. In fact, the main goal of any resource reservation algorithm is to provide each process Q units of budget every interval of time P. In the previous example, this is not true: from time 7 to time 17 the first process cannot execute.

This problem is partially solved by the GRUB algorithm. Consider again the previous example. In Figure 2.2 we show the schedule produced by the GRUB algorithm.

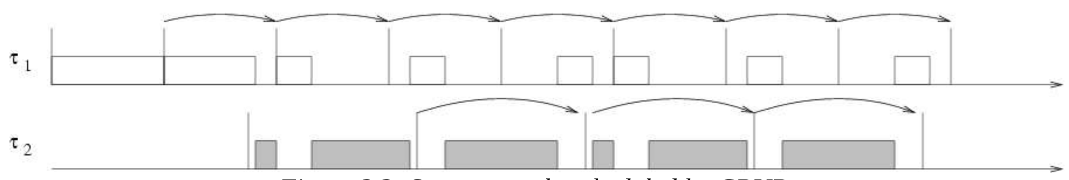

Figure 2.2: Same example scheduled by GRUB.

Algorithm GRUB maintains a variable U that keeps track of the total bandwidth used by the active processes. It uses this information to reclaim unused bandwith and give it entirely to the executing task (GRUB stands for *greedy reclamation of unused bandwidth*). Therefore, in interval [0,7] when the first process is the only active process, there is not unnecessary postponing of the server deadline. For this reason, when the second process arrives, the deadline of the first server is not too far and the deadline aging problem does not happen.

However, GRUB suffers from another undesirable problem that is in some way related to the previous one. Consider The following example.

Example 2. Consider a system scheduled by the GRUB algorithm consisting of a process that is always active and is served by a server with budget $Q_1 = 1$ and period $P_1 = 4$. Another process is a periodic process with period $T_2 = 16$ that is served by a CBS with budget $Q_2 = 12$ and $P_2 = 16$. The resulting schedule is shown in Figure 2.3.

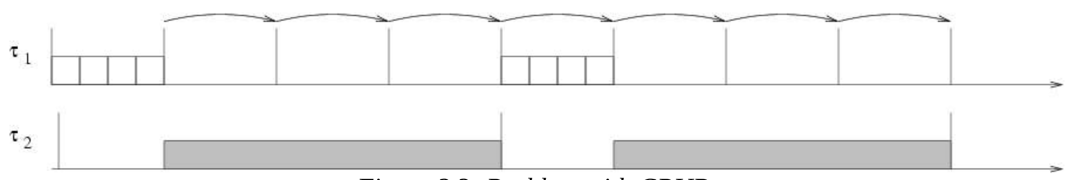

Figure 2.3: Problem with GRUB.

As you can see, the first process is not scheduled as we expect. In particular, it executes as it were served by a server with budget $Q' = 4$ and period $P' = 16$. Notice also that this behavior depends by the parameters of the other servers in the system. For example, if the second server had a period of $P = 20$, the first process would be scheduled as it were served by a server with budget $Q' = 5$ and $P' = 20$.

It is quite clear that the problem is caused by the fact that when the budget is exhausted the process is not suspended, but is inserted again in the ready queue with a new deadline.

We can solve this problem by introducing the concept of *hard reservation*. It was first introduced by Rajkumar [Raj97]. In a hard reservation, when the budget is exhausted, the process is suspended until the recharging time. In the case of CBS, a hard reservation can easily be implemented by suspending the process until the server deadline. We could do this by adding the following rule to the CBS:

Hard Reservation Rule: when the current budget q of the server is 0, the task is suspended until the current server deadline d . When the time is equal to the server deadline, the budget is recharged to $q = Q$, and the deadline is set to $d = d + P$.

As an example, we apply the previous rule to Example 2. The resulting schedule is shown in Figure 2.4.

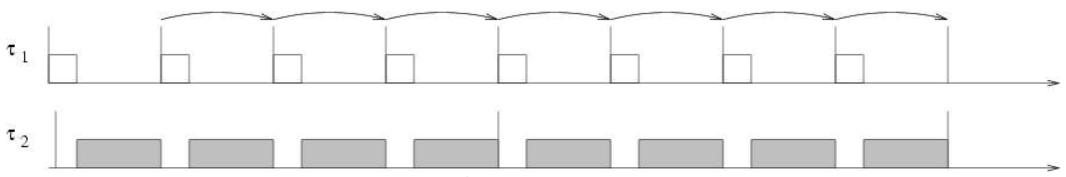

Figure 2.4: Schedule of Example 2 with Hard Reservations.

As you can see, the problem is now solved because we *forced* the first process to be executed inside its period.

However, even after introducing the Hard Reservation rule, there is still a small problem that needs to be addressed. It can happen that, in certain cases, the system becomes idle even if there is some process that needs to be executed. In fact, it can happen that some process finishes before we expect, while all other processes are suspended waiting for the recharging time.

Example 3. Consider again the system of Example 2, and suppose that the second process need to execute only 9.1 units of time. The resulting schedule is shown in Figure 2.5 .

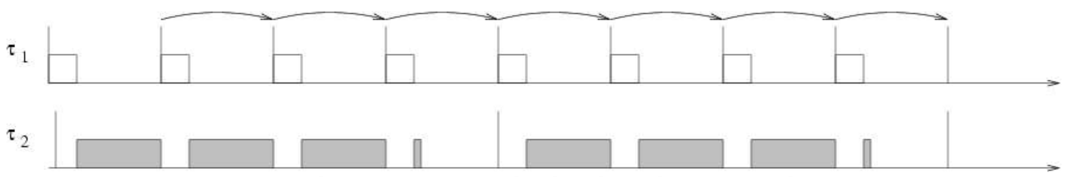

Figure 2.5: Hard reservartions make the algorithm non-work-conserving.

Althought the first process is always active, at time $t=13.1$ the system becomes idle. The first process is waiting for recharging while the second process has finished its instance. It is not easy to understand what to do. One possibility would be to recharge the budget of process 1 immediately. However, this solution can work in this example, but it is much more difficult to understand what to do when we have many processes in the system.

The CBS and GRUB algorithms were not designed for providing hard reservations and the previous case cannot be handled easily. This last problem is quite important in soft real-time system, where one of the main goals is to optimise the system resources. Therefore, we should use work-conserving algorithms.

Next, we describe a new scheduling algorithm, based on CBS, that solves all the three problems described in this section.

2.1 A new scheduling algorithm

In this section we describe a new server algorithm. Let's first introduce some definitions.

- Like a CBS server, our server has two parameters, the budget Q and the period P.

- The sum of all server bandwidths cannot exceed 1:

$$\sum_{i=1}^{n} \frac{Q_i}{P_i} \leqslant 1$$

- Every server has two internal variables: the current budget q and the current absolute deadline d. Our algorithm will perform the Earliest Deadline First algorithm among servers.

- Every server can have four states:

  a) **Inactive**, i.e. the server has no pending job and it does not contribute to the total bandwidth of the system

  b) **ActiveContending**,, i.e. the server has pending jobs and its current budget is greater than 0.

  c) **ActiveNonContending**, i.e. the server has no pending jobs but its bandwidth still contributes to the total system bandwidth.

  d) **Recharging**, i.e. the server has a pending job but its current budget is 0 and it has to wait to be recharged

- The system maintains

  a) a ready queue, where all ActiveContending servers are ordered by deadline

  b) a recharging queue, where all Recharging servers are ordered by recharging time

  c) a suspended queue, where all ActiveNonContending servers are ordered by inactive time.

  d) a total system bandwidth $U(t)$ that is the sum of the bandwidths of all the servers that are not in the Inactive state.

The state diagram for the algorithm is shown in Figure 2.6 The servers change state according to the following rules:

1. Initially all servers are in the Inactive state

2. If a job arrives at time t

   a) If the server is Inactive, then $q = Q$ and $d = t + P$

   b) if the server is ActiveContending or Recharging, the arrival is buffered and will be served later

   c) if the server is ActiveNonContending, it becomes active contending and it is inserted again in the ready queue with the same current budget and deadline

3. When the server executes for δ , $q = q - \delta$

4. If the server is ActiveContending and $q = 0$, the server reaches the Recharging

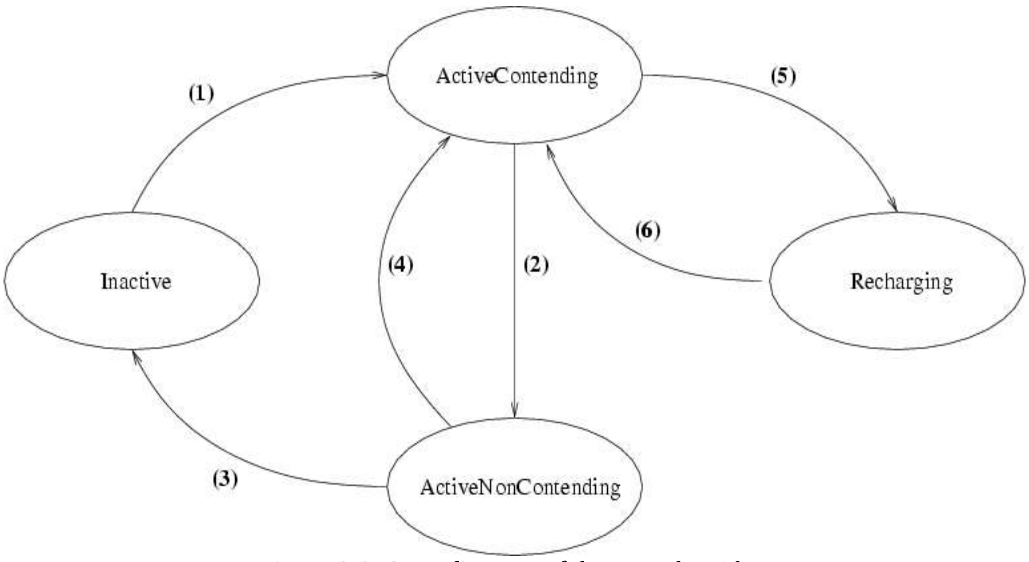

Figure 2.6: State diagram of the new algorithm.

state and the recharging time is set to $r = d$.

5. If the server is in the Recharging state and $t = r$, then the server become ActiveContending and $q = Q$ and $d = d + P$.
6. When the job finishes
   a) If there is another pending job, the server remains in the ActiveContending state.
   b) If there are no pending jobs and $t \geqslant d - q\frac{Q}{P}$, the server becomes inactive
   c) otherwise, the server becomes ActiveNonContending and the inactive time is set to $i = d - q\frac{Q}{P}$.
7. If the server is ActiveNonContending and $t = i$, then the server becomes inactive.
8. If at time t no server is in ActiveContending state and there is at least one server in Recharging state:
   a) let j be the first server in the recharging queue (i.e. The one with the smallest recharging time), and let $\delta = r_i - t$. For every server i in the recharging queue, $r_i = r_i - \delta$.
   b) Every server i in the recharging queue with $r_i = t$ is removed from the recharging queue and inserted in the ready queue; its budget is recharged to $q = Q$ and its deadline is set to $d_i = t + P_i$.

The way the algorithm works is better explained by an example.

Example 4. Consider a system consisting of 3 tasks, τ_1 , τ_2 and τ_3 . Task

τ_1 is always active and is assigned a server with budget $Q_1 = 1$ and period $P_1 = 4$. Task τ_2 is a periodic real time task with computation time $C_2 = 1$ and period $T_2 = 6$. It is assigned a server with $Q_2 = 2$ and $P_2 = 6$. Task τ_3 is always active and it is assigned a sever with $Q_3 = 2$ and $P_3 = 9$.

The system is underutilised, because the sum of the bandwidths of all server is less than 1. Moreover, τ_2 uses less bandwidth than expected (only 1 unit whereas it is allocated 2), and we would like to reclaim this exceeding bandwidth to execute the other two tasks.

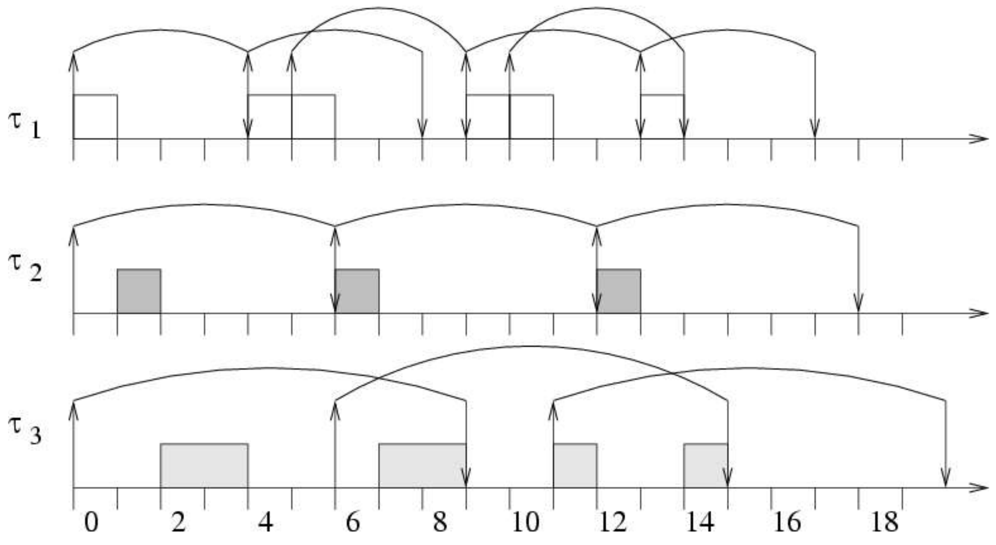

Figure 2.7: Example of schedule with the new algorithm

The resulting schedule is shown in Figure 2.7. Each activation of a server is represented with an upward arrow and the corresponding deadline is represented with a downward arrow. The activation instant and the corresponding deadline are linked by an arc. For example, task τ_3 is activated at time 0 with deadline at 9, at time 6 with deadline at 15 and at time 11 with deadline at 20.

Let's now analyse the schedule.

- At time 0 all tasks are ready, and task τ_1 is the one with the earliest deadline and execute. At time 1, the budget is 0 and the server goes to the Recharging state. The server is inserted in the recharging queue with recharging time $r_1 = 4$.
- Then tasks τ_2 is executed and finish its execution at time 2 without exhausting its budget. It goes to the ActiveNonContending state, with inactive time at 3.
- Task τ_3 is executed and its budget goes to 0. Like task τ_1, it is inserted in the recharging queue with recharging time $r_3 = 9$. Meanwhile, task %tau%_2 is now in the Inactive state.
- At time 4, the recharging time for task τ_1 is come, so it is put in the ActiveContending state and its budget is recharged to $q_1 = 1$ and its deadline is set

to $d_1 = 8$.

- Task τ_1 is selected to execute, and its budget become 0 again at time 5. It is then put again in the Recharging with $r_1 = 8$. Note that until now the schedule is the same as with CBS.
- At time 5, there is not other task in ActiveContending. Therefore, rule 8 is applied. The earliest recharging time is $r_1 = 8$. Therefore, all recharging times are decremented by $\delta = r_1 - t = 3$, with the result that $r_1 = 5$ and $r_3 = 6$. Now, task τ_1 is put again in the ready queue with budget recharged to $q_1 = 1$ and deadline set at $d_1 = 9$.
- Since it is the only task in the ready queue, τ_1 executes and again exhaust its budget and is put again in Recharging with $r_1 = 9$
- At time 6, task τ_2 arrives and needs to be executed again with deadline $d_2 = 12$. Moreover, task τ_3 recharging time is arrived and it is put in the ready queue with budget $q_3 = 2$ and deadline $d_3 = 15$

The interested reader can go through the remaining of the schedule to check how the algorithm works. A few things need to be highlighted. Since task τ_1 and τ_3 are always active and never suspend themselves, they spend their time between state ActiveContending and Recharging. Task τ_2 is using less than expected, therefore it goes through states Inactive, ActiveContending and ActiveNonContending. Note that there are no idle times, as expected, since there are two tasks that are always active. Note also that each task executes *at least* Q units of budget every P.

2.2Implementation in OCERA

The proposed algorithm is still in the stage of definition. We need to prove that the algorithm provides at least the same good properties of CBS and GRUB and, in addition, that it solves the problems presented in the previous section. Also, we would like to compare the proposed algorithm with other possible approaches. Finally, we will implement the algorithm in the next version of the QRES component.

3 Feedback Scheduler

Before describing the work that will be done in the next phase of the OCERA project for what concerns the Resource Manager component, we briefly recall the basic ideas underlying the current version of the feedback controller.

3.1 The feedback controller

The algorithm is based on the assumption that that underlying scheduler is a reservation based scheduler, such the CBS [Abe98]. The use of this class of schedulers is very important because

- it permits to analyse and control each task separately from the rest of the system. In fact, a reservation based scheduler provides *temporal protection*, i.e. to some extent the temporal behaviour of one task is not influenced by the temporal behaviour of the other tasks,
- and the temporal behaviour of the task can be precisely modeled.

The second property is very important because it allows us to describe precisely the mathematical relation between the inputs of the system, its internal state and its outputs. To the best of our knolwedge, our approach is the first one that uses an *exact* model of the scheduler. We now briefly recall this model.

3.2 Scheduler model

A periodic task τ_i is a (possibly infinite) sequence of jobs $J_i(1), J_i(2), \ldots$, each one characterised by an arrival time $a_i(k) = k T_i$, a deadline $d_i(k) = (k+1) T_i$ and a computation time $c_i(k)$. The system consists of one or more soft real-time periodic tasks. The goal is to minimise the scheduling error for each task. The scheduling error is defined as the difference between the finishing time of the job and its deadline:

$$\epsilon_i(k+1) = f_i(k) - d_i(k)$$

In the general case, the finishing time is very much influenced by the scheduling algorithm. However, a remarkable simplification in this respect arise from using a reservation based scheduler. The approach can be described as follows.

Each task is served by a CBS server with budget $Q_i(k)$ and period P_i . We define the server bandwidth $U_i(k)$ as $Q_i(k)/P_i$. Note that the budget and the bandwidth depends on k, because our feedback controller will modify the server bandwidth according to its rule.

For any reservation based system (also for the CBS), we can define the *virtual finishing time* $VFT_i(k)$ of job $J_i(k)$ as the time it would finish in a dedicated processor of speed $U_i(k)$. For example, if job $J_i(k)$ arrives at time $a_i(k)$ and requests $c_i(k)$ units of computation time, it would finish at time

$r_i(k)+\frac{c_i(k)}{U_i(k)}$ in the dedicated slower processor. Therefore, its

virtual finishing time is

$$VFT_i(k)=\frac{c_i(k)}{U_i(k)} \qquad (1)$$

Intuitively, if $VFT_i(k)>P_i$, then we need to allocate a larger reservation to task τ_i , (i.e. we need to speed-up the dedicated processor) in order to fulfil its requirements.

In any reservation based scheduler, there is a relationship between the virtual finishing time and the finishing time. By using the CBS with hard reservations, it can be proved [Lip00] that:

$$VFT_i(k)-P_i\leqslant f_i(k)\leqslant VFT_i(k)+P_i$$

Therefore, the smaller is P_i , the closest is the difference between the virtual finishing time and the actual finishing time. Since the virtual finishing time does not depend on the scheduling algorithm, or on the presence of other tasks in the system, we define our scheduling error on the virtual time:

$$\epsilon_i(k+1)=VFT_i(k)-d_i(k) \qquad (2)$$

If the scheduling error is greater than 0, then we are giving the task less bandwidth than necessary. If it is less than 0, then we are giving the task more bandwidth than necessary.

The objective of our controller is to keep the scheduling error as close as possible to 0. To ensure a good quality of service it is only required that the scheduling error not to be positive: however, we would also like to minimise the bandwidth given to the task, because in this way we can use the spare bandwidth for other purposes (for example, for other tasks or for other less important activities).

Now, Equation (1) is valid only when the server queue is empty when a job arrives. If a job arrives while another job is being served, this job is enqueued and it will be executed after the end of the current job. Therefore, the complete equation for the virtual finishing time can be written as follows:

$$VFT_i(k)=\left\{\begin{matrix} \frac{c_i(k)}{U_i(k)} & if\ VFT_i(k-1)\leqslant P_i \\ VFT_i(k-1)+\frac{c_i(k)}{U_i(k)} & if\ VFT_i(k-1)>P_i \end{matrix}\right\} \qquad (3)$$

and, the equation for the scheduling error is:

$$\epsilon_i(k+1)=\left\{\begin{array}{ll}\dfrac{c_i(k)}{U_i(k)}-P_i & if\ \epsilon_i(k)\leqslant 0\\ \epsilon_i(k)+\dfrac{c_i(k)}{U_i(k)}-P_i & if\ \epsilon_i(k)>0\end{array}\right\}$$

We further divide by P_i and introduce the step function $s_i(k)$ that is 1 when the scheduling error is positive and 0 otherwise. We also eliminate the subscript i because we are now going to analyse the behaviour of one task only. We finally set $u(k)=1/U(k)$. After all these substitutions, the equation for the scheduling error can be written as :

$$\epsilon(k)=s(k)\epsilon(k-1)+c(k)u(k)-1 \qquad (4)$$

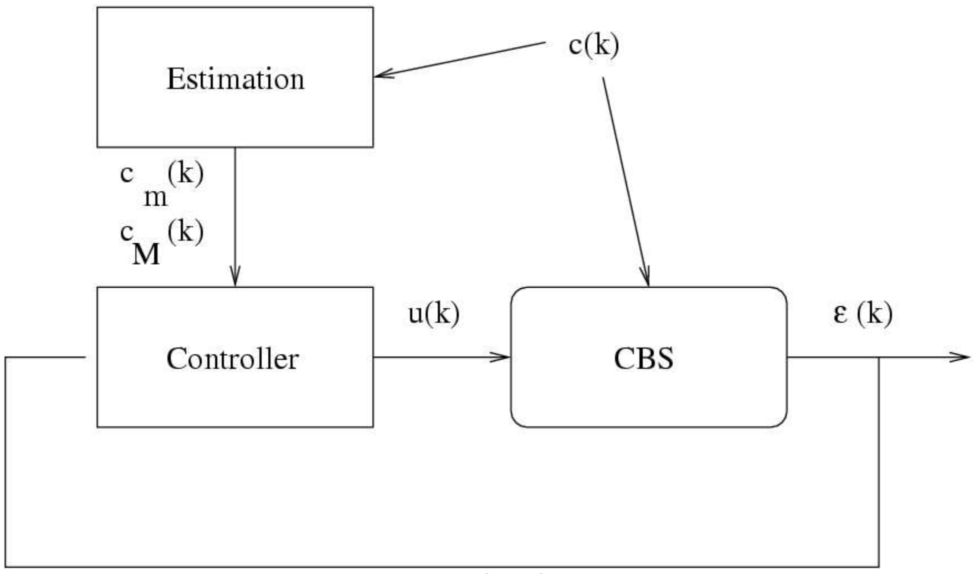

Figure 3.1: Structure of the feedback controller.

Note that:

1. the scheduling error is the output of the system that we want to control to a certain value. Tipically, we want to control the scheduling error to 0

2. $c(k)$ is an unknown (stochastic) variable. We can make some assumption on the possible evolution of this variable depending on the type of application we want to control. Namely, we assume that even though $c(k)$ is not known in advance, it is possible to estimate of an interval $[c_m(k), c_M(k)]$ where $c(k)$ will be contained with high probability.

3. $u(k)$ is the command variable. We can change this variable in order to control the scheduling error to 0. However, this variable is bounded too. Remember that the sum of the bandwidth of all servers in the system must be less than 1.

The second observation is especially noteworthy. In the current version of the feedback controller we made very general assumptions on the evolution of $c(k)$. This is useful because we can apply the same algorithm in different contexts.

The resulting controller scheme is depicted in Fig.3.1: the command variable $u(k)$ is decided based both on the past history of $\epsilon(k)$ and on the prediction of the future values for $c(k)$ performed by the estimation block. The proposed scheme has the following features:

1. The scheduling error remains bounded in a given interval provided that $c(k)$ complies with the estimated region.

2. The above region becomes more and more tight when a better estimation is used for $c(k)$.

3. The produced values for u(k) respect the saturation constraint on the bandwidth.

4. If at a certain time, the schedulign error is outside of the specified region, then the algorithm ensures convergence in a finite number of steps.

We now show some simulation that demonstrate the effectiveness of the approach. We used a real data stream (courtesy of Philips Research) taken from the decoding of a DVD stream. In Figure 3.2 we plot the decoding time for the first section of the stream.

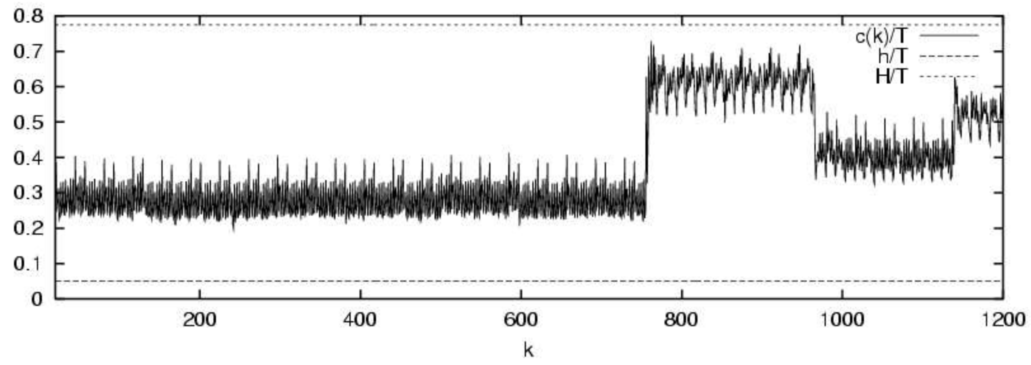

Figure 3.2: Decoding time for the DVD stream, normalised to the period (courtesy of Philips Research).

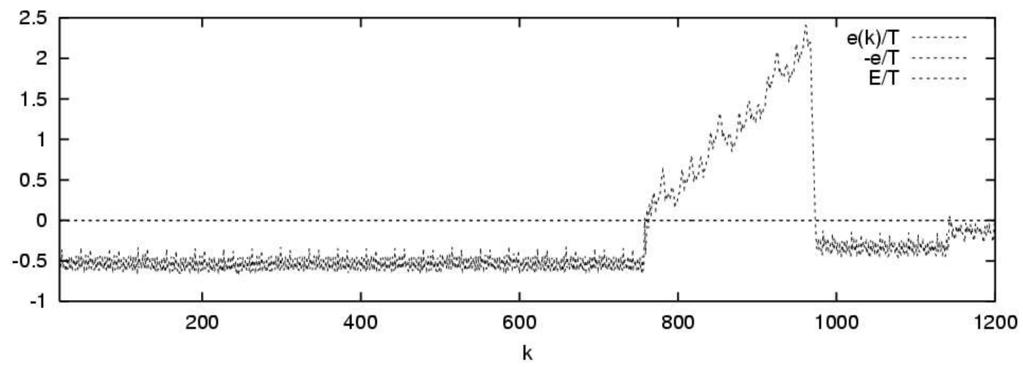

Figure 3.3: Scheduling error in case of static allocation of the bandwidth. The allocated bandwidth is slightly greater than the average required bandwidth.

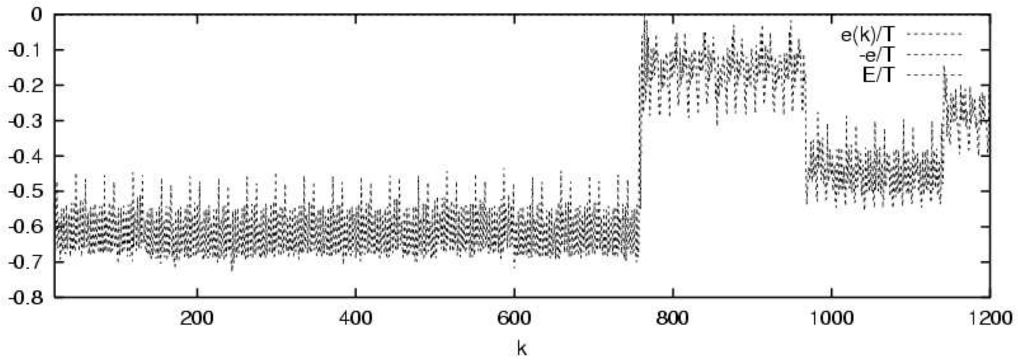

Figure 3.4: Scheduling error in case of static allocation of the bandwidth. The allocated bandwidth is equal to the maximum required bandwidth. There are large intervals over which the bandwidth is overallocated.

In Figures 3.3 and 3.4 we show the the scheduling error when we assign the application a fixed bandwidth and schedule it with CBS. As you can see, if we allocate the bandwidth based on the average value of $c(k)$ (Figure 3.3), the scheduling error can become very high. If we allocate the bandwidth on the maximum value of $c(k)$, we have an overallocation (the scheduling error is always lower than 0), wasting bandwidth.

In Figure 3.4 we show the scheduling error when the feedback control is in place. As you can see, the scheduling error is almost always inside [-0.2,0.2]. However, there is still space for improvement.

3.3 Improvements of the algorithm

In many practical cases $c(k)$ evolves according to some structure. For example, a MPEG2 stream is composed of three different kinds of frames: the ***I*** frames contain a lot of information and it takes more time to decode them; the ***P*** and ***B*** frames contain only partial information and can be decoded in a shorter time. Moreover, the sequence of frames in the stream often presents a periodic pattern. In our experiments, the pattern was ***IBBPBBPBBPBB***. Therefore, we can exploit this regularity in the estimation block (see Figure 3.1).

In particular, we ran some preliminary simulation of a new control algorithm based on more than one moving avarage. In Figure 3.5, the probability distribution functions for the scheduling errors are shown in the case of 1 moving average (i.e. the current implementation of the feedback controller), of 3 moving averages (i.e. it does not distinguish between I and P frames) and of 12 moving averages (i.e. it distinguishes between every single frame in the pattern). As it is possible to see, the last case is the most favourable.

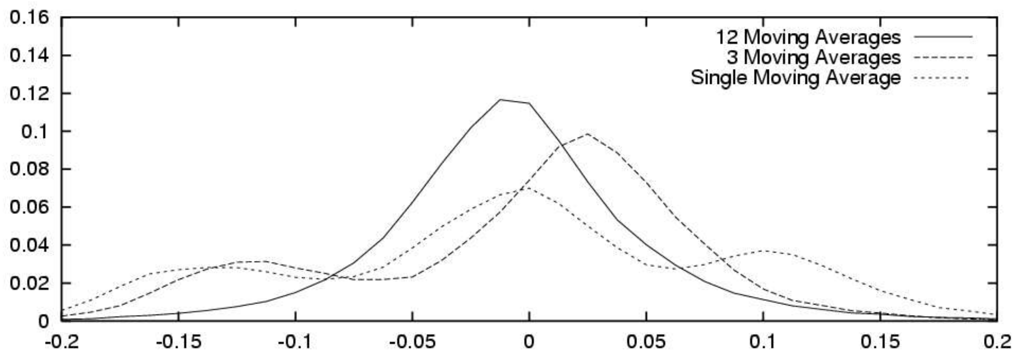

Figure 3.5: PDF of the scheduling error with different moving averages.

Bibliography

Abe02: Luca Abeni, Luigi Palopoli, Giuseppe Lipari, Jonathan Walpole, Analysis of a Reservation-Based Feedback Scheduler,Proc. of IEEE Real-time systems symposium, 2002

Abe98: Luca Abeni and Giorgio Buttazzo, Integrating Multimedia Applications in Hard Real-Time Systems,Proceeding of the 19th Real-Time Systems Symposium, 1998

Lip00: Giuseppe Lipari and Sanjoy Baruah, Greedy Reclamation of Unused Bandwidth in Constant Bandwidth Servers,Euromicro Conference on Real-Time Systems, 2000

Pal03: Luigi Palopoli, Luca Abeni, Giuseppe Lipari, On the applications of hybrid control to CPU Reservations,Proc. of Hybrid system computation and control, 2003