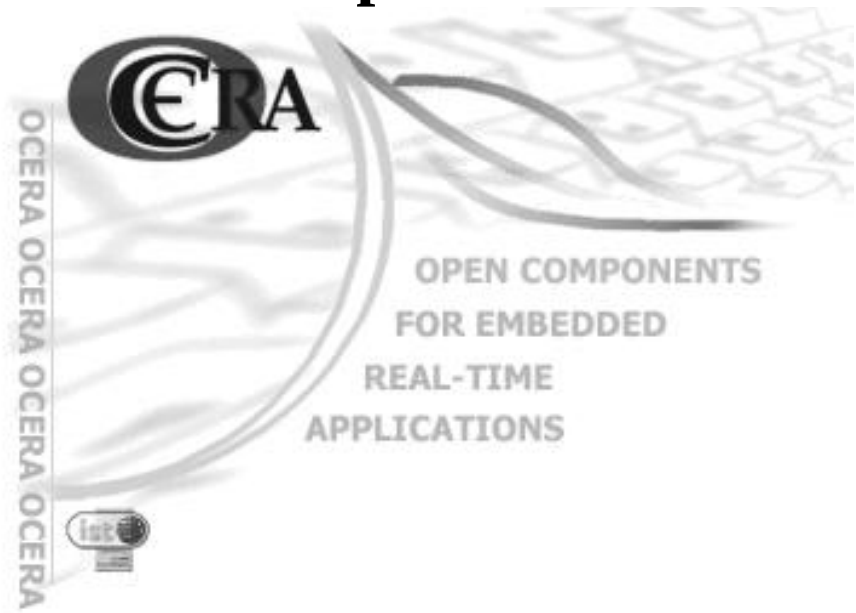
WP4 – Resource Management Components



Deliverable D4.4
Resource Management Components - V2

WP4 – Resource Management Components : Deliverable 4.4 – Resource Management Components V2
by Giuseppe Lipari, Luigi Palopoli, Luca Marzario, Tommaso Cucinotta

Published February 2004

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

In the last phase of Workpackage 4, we have refined the implementation of some QoS components and in particular:

- a new resource reservation algorithm, called IRIS, that improves over the CBS algorithm. It will be described in Chapter 2.
- a new module called *QoS Supervisor* that performs an admission control. In this way we separated the issue of admission control from the problem of scheduling, so that we can implement different admission control policies. Again, this is described in Chapter 2.
- A set of new *feedback scheduling* policies. They improve the behavior of the algorithms presented in Deliverable 4.2, and they are described in Chapter 3.

Other components, like the QoS library and the QoS monitor have not been modified since last version. In the following section, we summarize the architecture of the new QoS management modules in the OCERA kernel.

1.1Structure of the new QoS components

The architecture of the QoS components is shown in Figure 1.1 . All components are provided as dynamic loadable modules except the QMGR modules which can be provided as libraries or as modules.

The QSPV module is the *QoS Supervisor* that provides the admission control policy, whereas a set of QoS management modules implement the QoS control techniques described above.

This orthogonal separation among the different components is very useful because it permits to change ``on-the-fly'' the behaviour of any of the components and try out different feedback control architectures. In particular, the CPU allocation *mechanism* (the resource reservation scheduler) is separated from the policy implemented by the QoS modules. Moreover, both the scheduler and the controller are not embedded in the operating system and they can be dynamically changed at run-time by simply inserting/removing kernel modules. Finally, it is possible to have at the same time two or more control algorithms running into the same system, each one serving a group of different tasks.

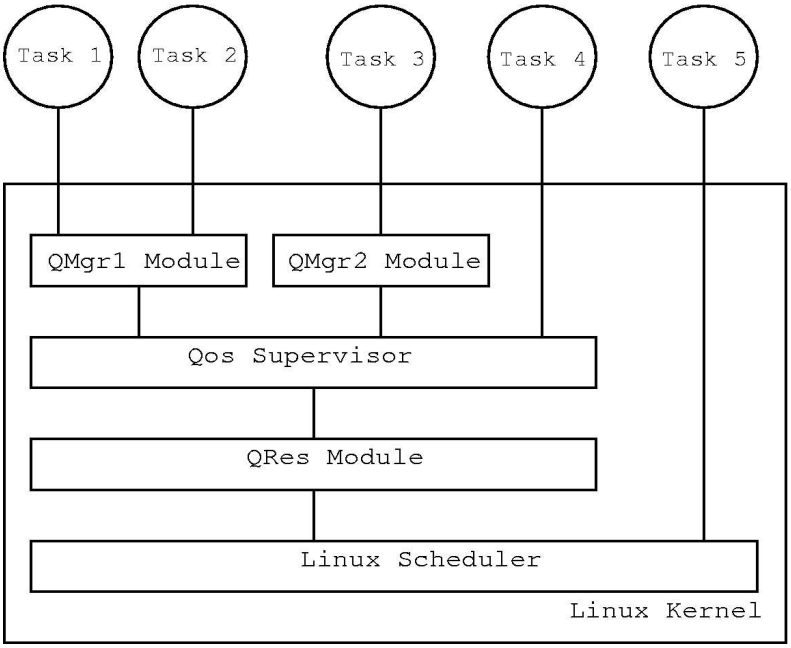


Figure 1.1 Architecture of QoS components

Scheduling module

The QRES module provides resource reservation scheduling. The resource reservation algorithm can be one between the CBS [Abe98], the GRUB [Lip00] or the IRIS [mar04]. The first two algorithm have been described in Deliverable 4.2, whereas the IRIS algorithm will be described in Chapter 2. It is possible to configure the QRES module to implement one of the above algorithm at compile time.

QoS Supervisor module

When a task is activated in the system requiring a certain bandwidth U_i , an admission test must be performed. The QoS supervisor module implements different admission policies depending on the user needs.

The QoS Manager module

We provide different QoS management modules (denoted with QMGR1 and QMGR2 in figure 1.1) that can coexist in the same system. Each module provides a different controller strategy and can serve more than one task. For a description of the possible controller strategies, please see Chapter 3.

2 The IRIS algorithm

During the first phase of the project, we 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 to give more bandwidth to processes that need to execute more and 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. The problem has been described in Deliverable 4.3. We also proposed a possible solution, by sketching the IRIS algorithm, a new bandwidth reservation algorithm that has been designed to purposely solve the deadline aging problem. In this chapter we present the complete description of the IRIS algorithm.

2.1 Description of the problem

The problem under investigation was presented in deliverable D4.3. In this section we summarize the problem for completeness of the report.

The Constant Bandwidth Server presents some problems when serving a non-periodic process that consist of one single instance that runs indefinitely. For example, in Figure 2.1 we show the schedule generated by the CBS when serving two non periodic tasks τ_1 and τ_2. Task τ_2 arrives in the system after some time, when the deadline of the first server is already far away. As a consequence, task τ_1 cannot execute for a while. We refer to this problem as the *deadline aging problem.*

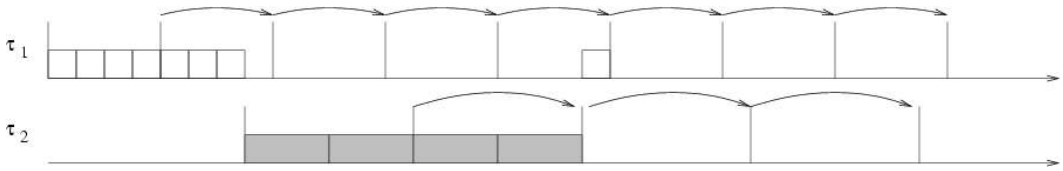


Figure 2.1: Problem with CBS.

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.

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.

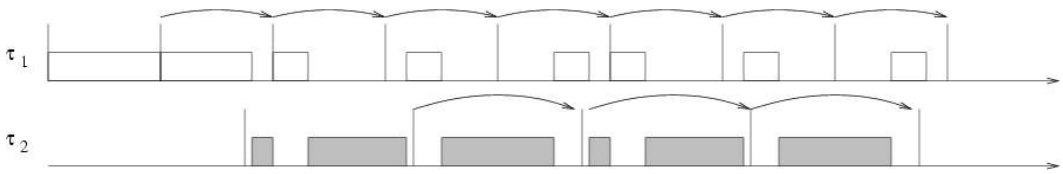


Figure 2.2: Same example scheduled by GRUB.

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.

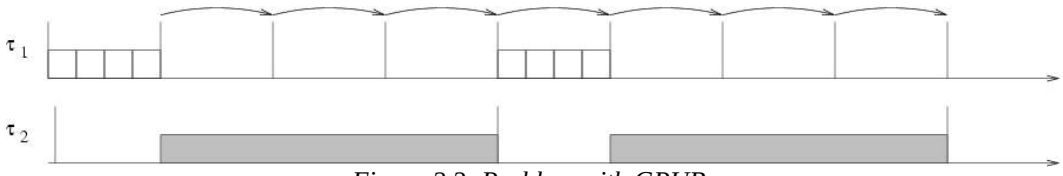


Figure 2.3: Problem with GRUB.

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.

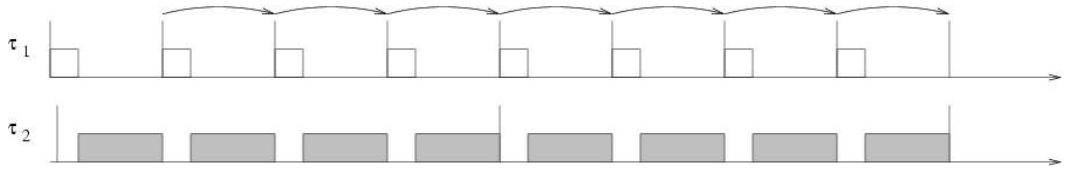


Figure 2.4: Schedule of Example 2 with Hard Reservations.

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 .

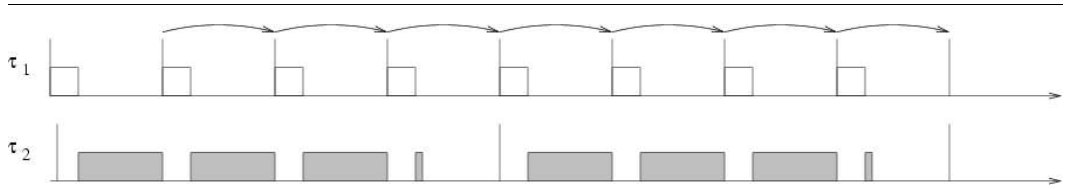


Figure 2.5: Hard reservartions make the algorithm non-work-conserving.

Although 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.

2.2 The IRIS algorithm

IRIS (Idle-time Reclaiming Improved Server) is a new scheduling algorithm that allows the coexistence of hard, soft and non real-time tasks. The proposed algorithm is specifically designed to handle computational overload. A task that needs more CPU-

time than reserved can re-use the spare bandwidth, without interfering with the others tasks. With respect to other reclamation schemes, the novelty of the proposed algorithm is that the spare bandwidth is fairly distributed among the needing servers. The effectiveness of the algorithm is demonstrated with an extensive set of experiments. We also propose a methodology to set scheduling parameters depending on the type of the task and on the time constraints needed.

System model

In this research, we consider two types of tasks in the system: cyclic tasks and non--cyclic tasks. A **cyclic task** τ_i consists of a main loop. At the end of every loop instance the task blocks waiting for some event. Each execution between two blockings is called *job* $J_{i,k}$. We indicate with C_i the worst-case execution time (WCET) of τ_i . Cyclic tasks can be divided into *periodic tasks* that wait for a periodic timer event with period T_i ; *sporadic tasks* that wait for external events with a minimum inter-arrival time, also called T_i ; *aperiodic tasks* that wait for external events with an unbounded inter-arrival time. Each job $J_{i,k}$ of a cyclic task τ_i has associated an arrival time $a_{i,k}$ and a deadline $d_{i,k}$. We denote with $U_{\tau_i}=C_i/T_i$ the utilization factor of task τ_i . Moreover, cyclic tasks can also be divided into two classes, depending on their criticality: hard real-time and soft real-time. For the hard real-time tasks, it must be guaranteed that every job completes before its deadline. Soft real-time task are less critical: deadline miss could result in a performance degradation but not in a critical fault. We assume that the relative deadline of cyclic tasks is equal to T_i .

Acyclic tasks model batch activities that are continuously active for large intervals of time, like for example the process of compiling a program, or a long scientific simulation or calculation, or a control tasks polling an external sensor. Usually, they are not associated any temporal constraint. However, in some case it may be desirable to execute them at a certain minimum rate. Therefore, the aim is to assign them a minimum fraction of the processor bandwidth.

A **server** is an abstract entity used by the scheduler to reserve a fraction of CPU-time to a task. Each server S_i is characterized by the following parameters: P_i is the period of the reservation and it characterizes the granularity of reservation; Q_i is the reserved execution time per period; U_i is the fraction of reserved CPU-time, also called utilization factor, and it is defined as $U_i=\frac{Q_i}{P_i}$. In addition, each server maintains its own internal variables that are updated by the scheduler depending on the server rules. One of these variables is the server priority.

The servers are inserted in the priority queue of the scheduler. When a server is selected by the scheduler because it has the highest priority, the corresponding task is executed. While a task is executing, the budget of the server is decremented accordingly. We denote by q_i the current budget of server S_i . When the budget get exhausted, the task is stopped until the budget is replenished. We denote with $r_{i,k}$ the time at which the budget is replenished. In this paper we consider only dynamic server: each server has a dynamic deadline d_i that is used by the scheduler to order tasks in the priority queue, accordingly with EDF algorithm . Different server algorithms use different

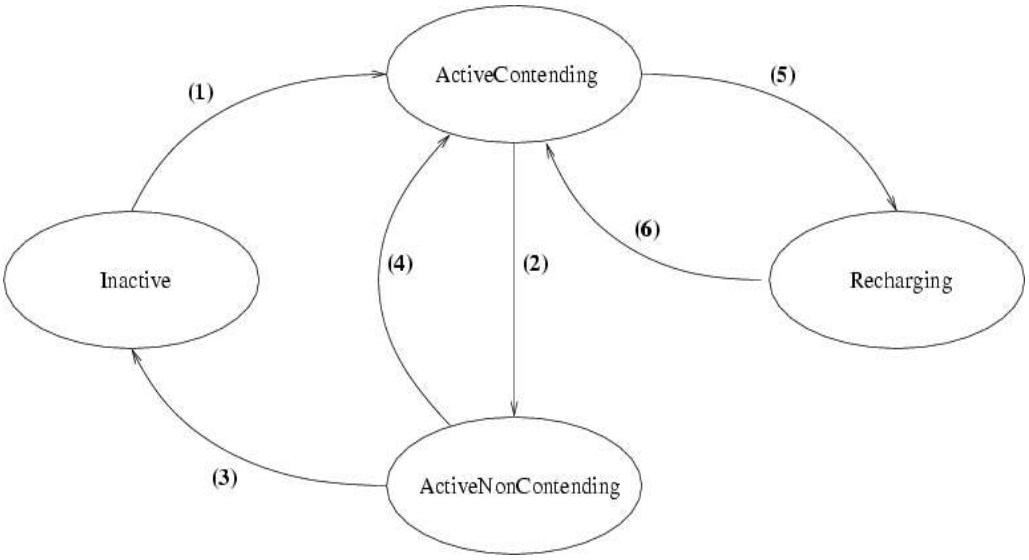


Figure 2.6: State diagram of the new algorithm.

policies for updating q_i and d_i .

The goal of our research is to design an algorithm that guarantees a minimum fraction of CPU time to each task (both cyclic and acyclic) in every interval of time of a given length. This will allow to independently analyze the temporal behavior of each task and to guarantee hard real-time tasks.

Description of the IRIS algorithm

In this section we describe IRIS.

- Every server can have four states (as shown in Figure 2.6):

  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 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 1. 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.

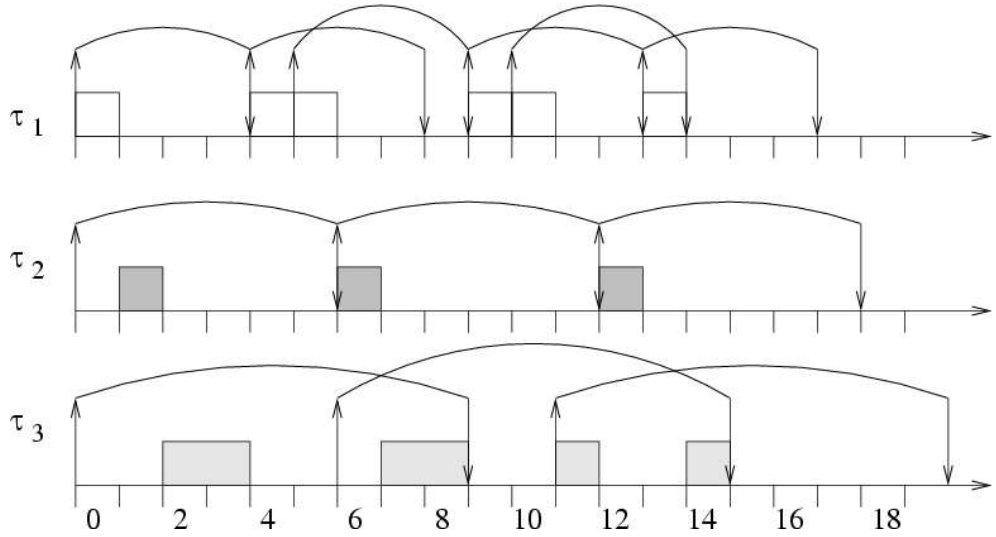


Figure 2.7: Example of schedule with the new algorithm

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 τ_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.

We also define a simplified version of IRIS, that we call IRIS-HR, for which the warping rule (8) is not applied. The IRIS-HR is useful in those case in which we want an upper bound on the amount of time assigned to one application, even in the case of spare bandwidth left.

Properties of the algorithm

The IRIS algorithm maintains the original CBS schedulability property. It is also a fair algorithm in the sense that the spare time is equally distributed among the servers that need to execute more than the reserved CPU time. The following Theorems are reported here without a proof. The proof of these theorems can be found in [mar04].

Theorem 1 (Schedulability Property)

Given a set of periodic tasks with total utilization factor $U_p=\sum_{i=1}^{N} U_{\tau_i}$ and a set of IRIS servers with utilization factor $U_s=\sum_{i=1}^{N} U_i$ the whole set is schedulable by EDF if and only if

$U_p+U_s\leqslant 1$.

Theorem 2 (Hard Schedulability)

A hard task τ_i with period T_i and WCET C_i is schedulable by an IRIS server with parameters $Q_i \geq C_i$ and $P_i = T_i$ if and only if τ_i is schedulable with EDF.

Theorem 3 (Minimal execution time guarantee)

Let $[t_1, t_2]$ be an interval such that server S_i, serving task τ_i, is never inactive. If $t_2 - t_1 \geq 2P_i - Q_i$, then τ_i executes at least Q_i unit of time in the interval $[t_1, t_2]$.

An interesting property of IRIS server is that it solves the deadline aging problem.

Theorem 4 (Deadline aging)

If d_i is the deadline associated to the server S_i at time t, we have that:

$$\forall i, t,\ d_i \leq t + P_i$$

The IRIS algorithm distributes the spare time among the reclaiming servers in a fair way: the spare time of the system is redistributed to the servers proportionally to the reserved bandwidths. Since the allocation of CPU time is not fluid, we show the property only in intervals large enough with respect to the periods of the servers.

For simplicity we analyze the case in which the served tasks are always ready to execute. We denote by $U_i'(t_1, t_2)$ the bandwidth used by server S_i in the interval $[t_1, t_2]$:

$$U_i'(t_1, t_2) = \frac{Q_i'(t_2) - Q_i'(t_1)}{t_2 - t_1},$$

where $Q_i'(t)$ is the amount of budget used by server S_i until time t.

Theorem 5 (Fairness)

Consider two IRIS servers S_i, S_j serving tasks that are always ready to execute (the servers are always in the state active or Re-charging). The following property holds:

$$\forall t_1, t_2:\ t_2 - t_1 \geq 2P_{max}$$

$$\frac{U_i}{U_j}\frac{(t_2 - t_1 - 2P_i)}{(t_2 - t_1 + 2P_j)} \leq \frac{U_i'(t_1, t_2)}{U_j'(t_1, t_2)} \leq \frac{U_i}{U_j}\frac{(t_2 - t_1 + 2P_i)}{(t_2 - t_1 - 2P_j)}$$

where $P_{max} = max(P_i, P_j)$.

How to assign IRIS parameters

All the properties mentioned in the previous section allow us to use a bandwidth reservation strategy to allocate a fraction of the CPU time to each task independently of

the other tasks in the system. In particular, we can execute in the same system different types of task, cyclic and acyclic, hard and soft. In fact, the reserved bandwidth, if available at the initial request, is always guaranteed. This guarantee is based on the reservation period and does not depend on the behavior of other servers.

In this section, we give some suggestions on how to assign the budget and the period to the different tasks in the the system.

- **Hard real-time tasks**. Hard real-time tasks can be scheduled directly by EDF or through a dedicated IRIS server. If we are absolutely sure about their WCET, then we can schedule them directly under EDF. However, if for some fault, a HRT task executes more than its computed WCET, some other random task in the system may miss its deadline. In the worst case, this could jeopardize the CPU. If we want to avoid this interference, we can assign each hard real-time task an IRIS server, setting P_i to the task's period T_i and Q_i greater than its estimated WCET. In this way, the other tasks in the system are protected by possible misbehaviors. This is useful especially in the developing and debugging phase.

- **Soft real-time tasks.** In the case of periodic soft real-time tasks, we can set P_i to the task's period. For aperiodic tasks, the server period should be set equal to the average response time needed. To set Q_i we have more freedom: since setting it to the WCET may waste the CPU utilization, we can set this parameter to some value between the task's average execution time and its WCET. The trade off is between the amount of reserved bandwidth and the number of deadline miss. Notice that, since IRIS is able to reclaim the spare bandwidth, it may happen that, even by setting the budget to a small value, the number of missed deadlines is low because of the extra budget dynamically available due to the reclaiming mechanism.

- **Non real-time tasks.** These tasks do not have time constraints. However, in many cases it could be desirable to execute them at a certain rate. In this case a server could be used to reserve a fraction of CPU time to each task. P_i must be set to the granularity of reservation, while Q_i must be set proportionally to the needed bandwidth. As an example, we can use a server to reserve a fraction of CPU time to an interactive task like a shell to control or to monitor the system.

Experiments

In this section, we show the effectiveness of the IRIS algorithm. The goal is to highlight two important characteristics of IRIS against CBS: good performance in overload situations, and minimal temporal utilization guarantee. A massive number of tests have been run using a synthetic workload model. In the following, we describe the simulations settings and the obtained results.

The task sets for these experiments were generated with the following characteristics:

- Three hard real-time periodic tasks, generated with random C_k and T_k . The bandwidth consumed by hard tasks is:

$$U_{hard} = \sum_{k=1}^{3} \frac{C_{\tau_k}}{P_{\tau_k}}$$

- Two soft tasks that serve sporadic jobs, whose inter-arrival and execution times are

also randomly generated around the values Q_i and P_i . A total of 300 jobs are generated for each task. The soft load is:

$$U_{soft} = \sum_{k=4}^{5} \frac{Q_k}{P_k}$$

The variable measured is the average soft response time of sporadic tasks, when task sets are scheduled under IRIS and CBS. Being $rt_{i,j}$ the response time of the job $J_{i,j}$, the average soft response time is defined as follows:

$$avsrt_i = \sum_{j=0}^{j=300} \frac{rt_{i,j}}{r_{i,j+1} - r_{i,j}}$$

In the first experiment, soft tasks consume 15% of the total utilization of the system ($U_{soft} = 0.15$) while hard task parameters are specified with U_{hard} ranging from 0.2 to 0.6. We assume that the computation time of the jobs ($C_{i,j}$) is similar to the budget of the server assigned to the task (Q_i). As it can be seen in Figure 2.8, average soft response time is very similar in IRIS and CBS, although it is slightly lower in the case of IRIS.

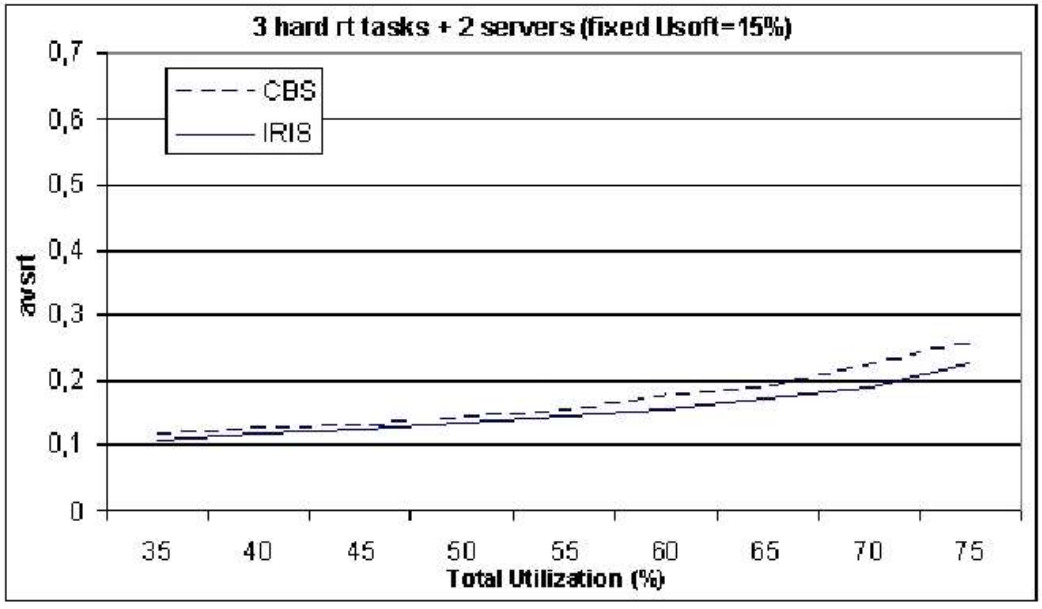


Figure 2.8 Experiment 1

Experiment 2 has been generated with the same characteristics than Experiment 1, except that an overload situation is forced. We can create an overloaded system if some jobs have a $C_{i,j}$ much greater than its budget (we call it "heavy" jobs). In the simulations, every 10 activations $C_{i,j} = 6Q_i$. The results are shown in Figure 2.9. In this case, IRIS performs much better than CBS, because CBS suffers from deadline aging, and when a "heavy" job arrives, the budget is replenished several times and the deadline is moved away. This way, the response time of the "heavy" job, and also the response time of future jobs, is highly increased. However, as IRIS does not replenish the budget

immediately, the jobs that arrive after the "heavy" one do not suffer the consequences of the overload, maintaining a reasonable response time.

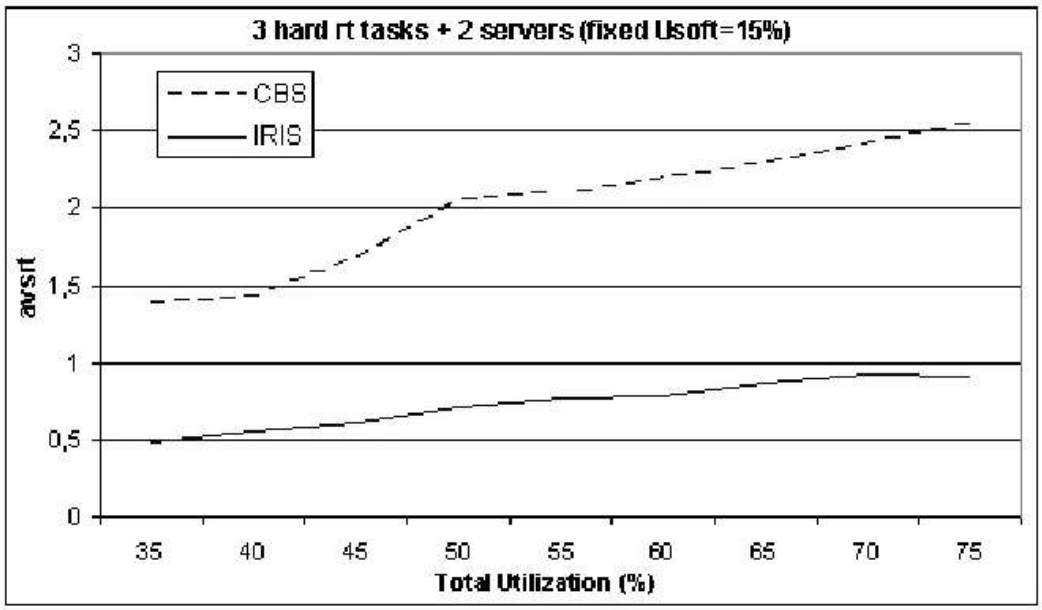


Figure 2.9 Experiment 2

In Experiments 3 and 4, hard tasks consumes 35% of the total utilization of the system ($U_{hard}=0.35$) while soft tasks parameters Q and P are generated with U_{soft} ranging from 0.1 to 0.5. Experiment 3 simulates a normal execution (Figure 2.10) while Experiment 4 shows the results in an overload situation (Figure 2.11).

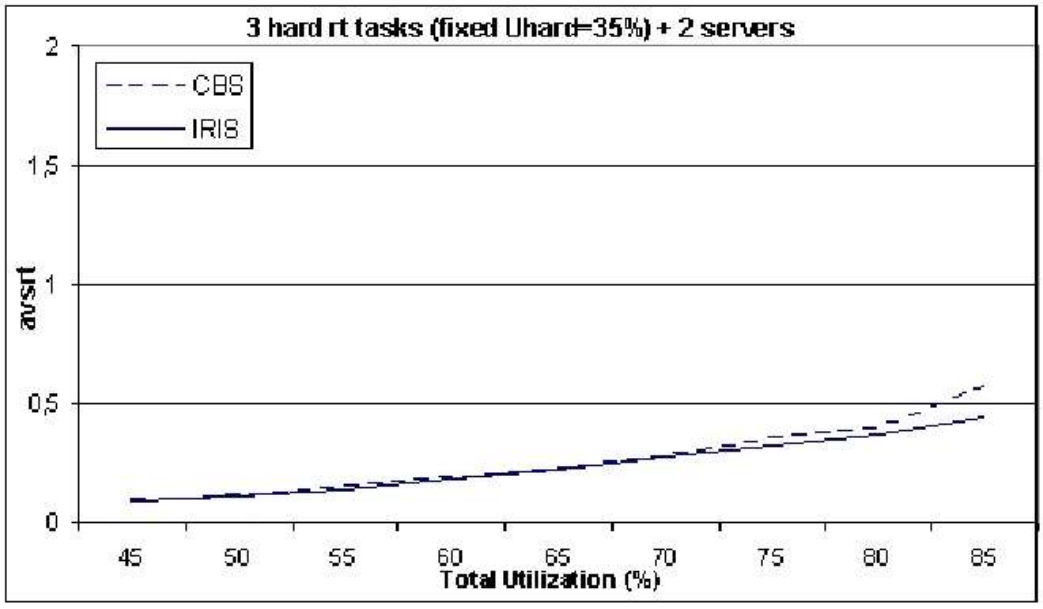


Figure 2.10 Experiment 3

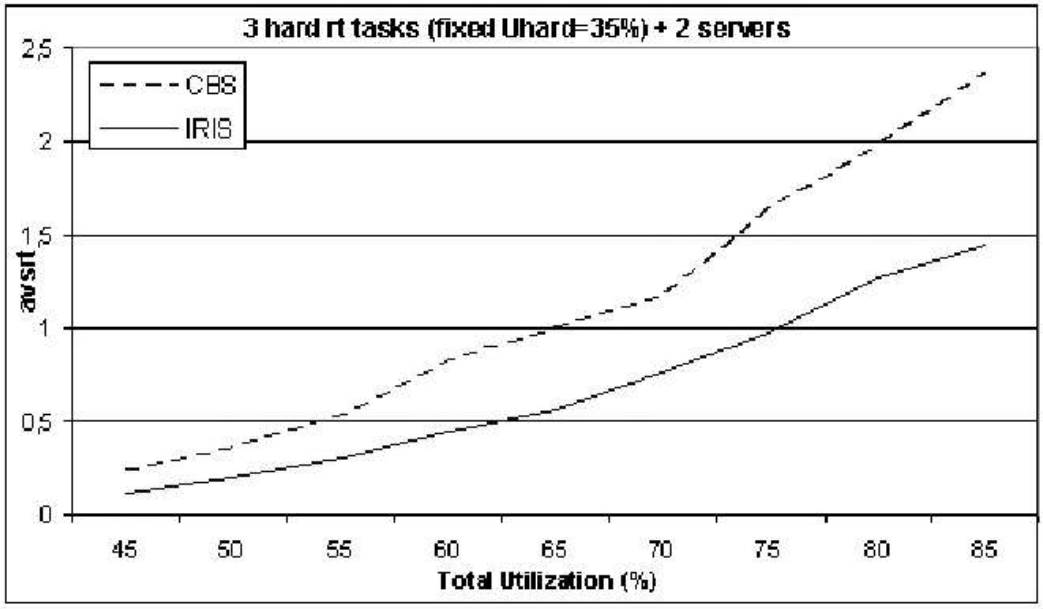


Figure 2.11 Experiment 4

In both experiments, the performance of IRIS is better than CBS, with a great improvement in the overloaded systems.

The number of missed tasks' deadlines is also an important parameter that must be taken into account. Figure 2.12 shows the percentage of missed deadlines with both algorithms for the task sets generated in Experiment 4. The results show that the maximum deadline misses of IRIS is less than 30%, being always greater in CBS.

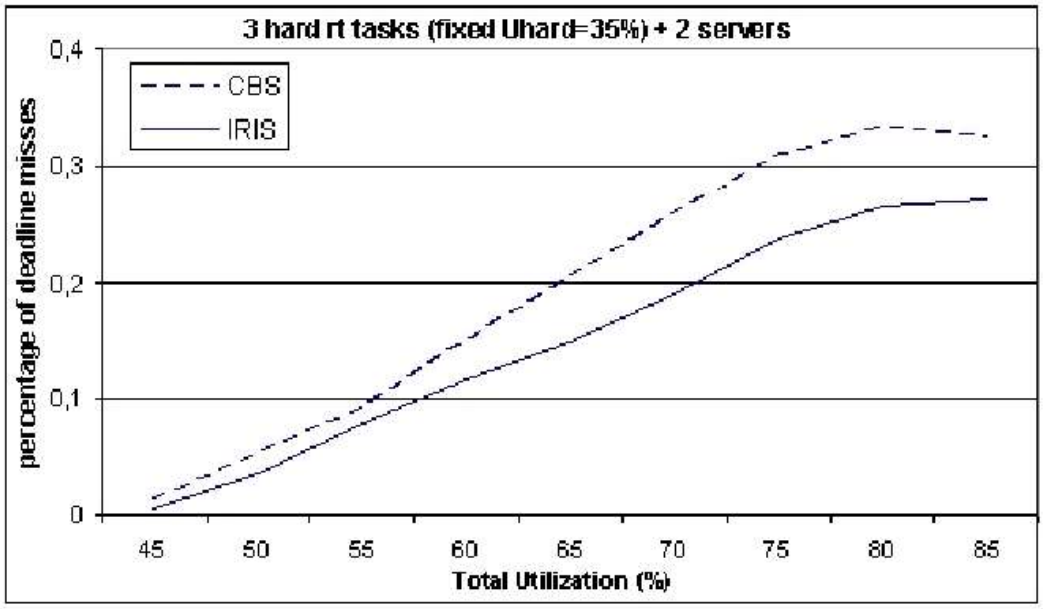


Figure 2.12 Experiment 5

Another interesting experiment is to compare the utilization of soft tasks with IRIS and

CBS. In the first experiment, the workload consists of 6 greedy tasks ($\tau_{1,..},\tau_6$), with Q_i and P_i randomly generated, except budget for τ_6 that is $Q_6=1$. Task set parameters are generated to have different utilization (from 20 to 80%).

In this case, the variable measured is the distance between two consecutive executions of task τ_6 . As this task has $Q_6=1$, we want to see if the required utilization of the task is guaranteed for both algorithms. During the simulations, the number of occurrences of every value of the distance have been counted. The distance between two consecutive executions has been normalized to the period P_6 . Figure 2.13 shows the results.

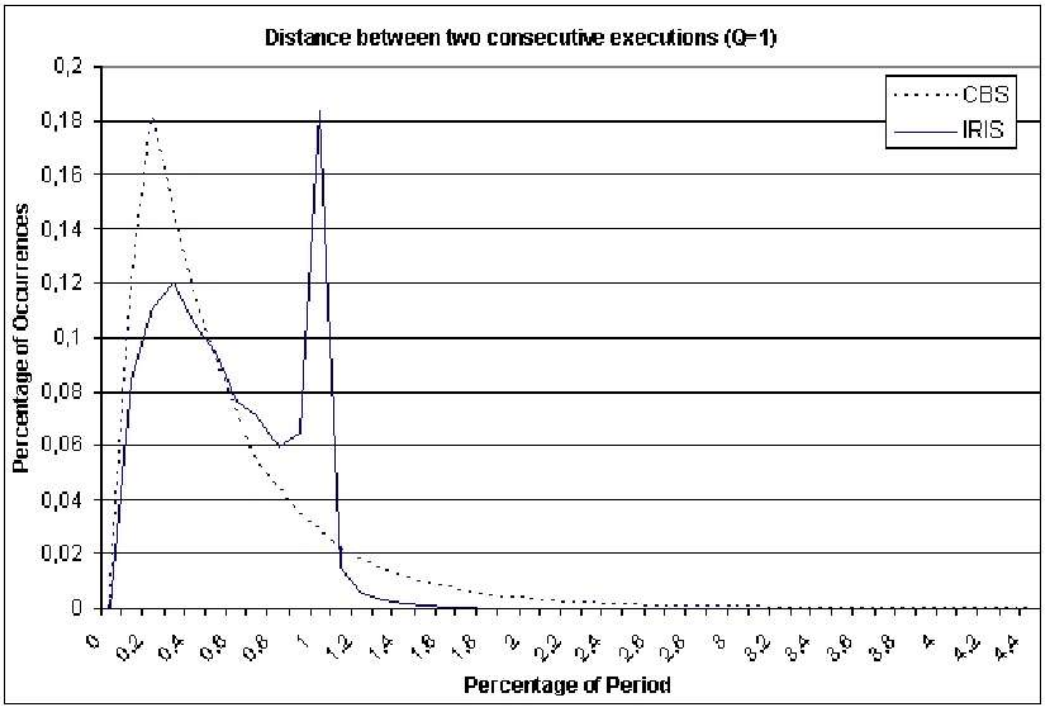


Figure 2.13 Experiment 6

As it can be seen, CBS executes with a great variability, and most of the times task executions are very close. But, due to deadline aging, when the budget is exhausted, the task scheduled under CBS can stay without executing for a long time. This is the reason why under CBS, the time between two consecutive executions can be up to 440% of the task period. However, a task scheduled under IRIS executes most of the times every period, or less. And the maximum interval between two executions is only 190% ($2P_6-Q_6$). This property of IRIS is specially important in systems where reducing jitter is a key issue, such in control systems.

The last experiment consists of measuring the utilization of greedy tasks all over the execution of the system, and see what happens to utilization when a new greedy task arrives. The goal is to demonstrate that IRIS is more fair in distributing the idle time than CBS.

Experiments have been done in the following way: three acyclic tasks τ_1 , τ_2 and τ_3 have been generated with server periods ranging from 10 to 30. First activation of

τ_1 and τ_2 is set to the initial instant (t=0), while first activation of τ_3 is set to instant t=30. The variable measured is $\frac{U_1'(t_1,t_2)}{U_1}$. Similar results have been obtained with $\frac{U_2'(t_1,t_2)}{U_2}$, but in Figure 2.14 are depicted results only for τ_1. Instants t_1 and t_2 have been chosen to measure utilization in windows that moves over the total execution of the system. The size of the windows is the largest period the three tasks. This means that, in Figure 2.14, point 1 in X coordinates means the first window ($t_1=0$, $t_2=maxperiod$), point 2 is the second window ($t_1=1$, $t_2=maxperiod+1$), and so on.

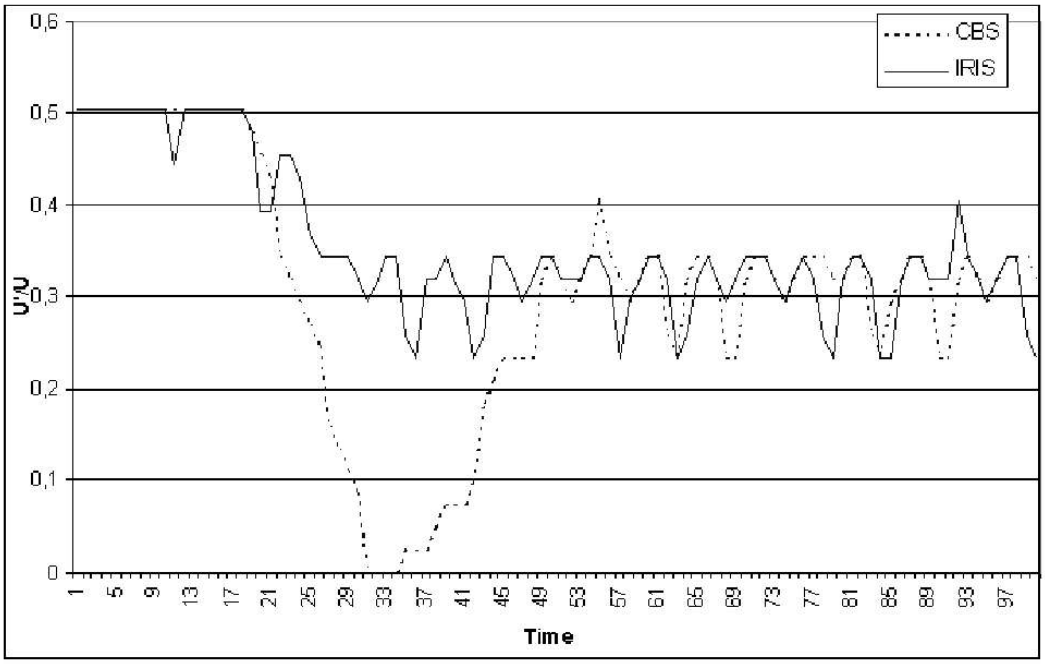


Figure 2.14 Experiment 7

Figure 2.14 shows that, in CBS, the utilization of τ_1 rapidly goes to 0 when τ_3 arrives (t=30). And the same happens to τ_2. As a consequence, CBS does not maintain the utilization required. However, IRIS maintains the required utilization of τ_1, even when new tasks arrives, or even in overload situations.

2.3 Implementation in OCERA

QoS supervisor

As anticipated in Chapter 1, we decided to separate the admission control from the scheduling algorithm. In this way, the user can customise the admission control policies to the needs of the applications more easily. This customisation is especially useful when introducing reservation with varying bandwidth, as in the case of feedback scheduling (see Chapter 3 for further details).

When a new reservation is created specifying a certain budget Q and a certain period P,

the system must check if there is enough free bandwidth to accommodate for the new reservation. This admission control is performed by the QoS supervisor module (denote with QSPV in Figure 1.1). This module intercepts all the calls to the scheduler_setsched () via the setsched hook (see deliverable D4.2 for an explanation of the scheduling hooks).

Three different flavors of this module exist, each one implementing a different admission control policy: *saturation*, *compression* and *reject*. They differ in their response to requests that cannot be accommodated. In all cases, if the sum of the CPU utilizations of the existing reservations, plus the utilization of the new reservation, does not exceed the maximum possible utilization U_{lub} , then the request is forwarded to the setsched handler of the QRES module; the sched _setscheduler() succeeds and the task will be scheduled according to the CBS algorithm with the specified parameters.

If there is not enough bandwidth to serve the new request, the action depends on the selected policy:

- In case the saturation policy is selected, the highest possible budget is assigned to the task so that the total CPU utilization does not exceed U_{lub} . The setsched handler of the QRES module is called with the new budget.

- In case the compression policy is selected, all the reservations are recomputed (``compressed'') so that we can make enough space for the new request. See [Abe99] [Abe02] for a detailed description of the compression algorithm. For each existing reservation, the setsched handler of the QRES module is invoked with the new budget value.

- In case the reject policy is specified, the sched_setscheduler() returns with error and the task is scheduled in background.

The QSPV module is also used by the QoS Manager to dynamically change the budget of an existing reservation according to the feedback control algorithm.

Implementation of IRIS

The implementation of the IRIS algorithm is very similar to the implementation of the CBS algorithm, described in Deliverable D4.1. The scheduler is implemented as a dynamically loadable module and requires that the "*generic sheduler patch"* is applied to the Linux kernel. We do not report here the complete description of the implementation and remand to Deliverable D4.1. Here, we highlight the differences between the implementation of the CBS and the implementation of the IRIS scheduler.

In IRIS, a new queue has been defined, the Recharging queue. It is an ordered list of server descriptors, which contains all servers that are in the Recharging state. The queue is ordered by increasing recharging time.

A system timer is programmed to the shortest between the following two events:

- The recharging time of the first server in the Recharging queue, if any. When this event expires, the first server is extracted from the Recharging queue, its current budget and deadline are recomputed and the server is inserted in the Ready queue;

- The budget exhaustion time of the executing server. When this event occurs, the executing server is suspended and inserted in the Recharging queue. A new server is

selected for execution from the Ready queue.

Overhead of the implementation

In order to measure the overhead introduced by the qres module in the scheduling process, we run a test that stress the scheduler, running several process simultaneously. In particular we run about 50 processes of various type (like terminals, browser, mailer etc.) served by the "Linux" server, that is the IRIS server that handles all processes that are not explicitly served by a dedicated server.

The bandwidth allocated to this server is 33% (budget 10 ms and period 30 ms). We run 30 processes, consisting in a X terminal executing the program "top" with the highest refresh frequency (100 Hz), with allocated 2% of total bandwidth to each process (budget 20 ms and period 1 sec). Hence, the total system bandwidth allocated is about 93% (Linux 33% + 30*2%). After an accurate analysis we realize that the most time-consuming functions are the stop() and dispatch() functions, that respectively schedule and deschedule all the tasks served by a server. The main reason is that they have to run a queue of all the tasks served by each server. This operation is O(n), so that the execution time increase with the number of task served. It is important to notice that the Linux scheduler (as of version 2.4.18) has the same problem, i.e. the complexity in the worst case is O(n). The current version of Linux (2.6.x) changed the implementation of the scheduler and has a constant time complexity O(1).

The same problem happens when the system have to handle a lot of servers.
The current implementation of the EDF can be improved, since it is well-known that it is possible to implement an insertion in an EDF queue with a complexity of O(log n).

Following is a table in which we show the overhead introduced by each hook (time are expressed in nanoseconds) on a Athlon XP 1800+ Mhz with processor frequency of 1533 Mhz.

	Average (ns)	**Minimum(ns)**	**Maximum (ns)**
unblock_hook (activation)	4260	21	42547
block_hook (deactivation)	5036	20	47497
fork_hook (creation)	869	161	2285
cleanup_hook (termination)	3647	314	45170
setsched_hook (scheduling policy selection)	79958	54312	99441

3 Feedback Scheduler

3.1 Motivation and background

The traditional way for using resource reservation scheduling is to reserve a fixed fraction of the CPU bandwidth to each task, so that its temporal constraints can be fulfilled.

However, a static allocation of resources is not a good idea if the task widely changes its execution requirements throughout its execution. Indeed, we can allocate the CPU bandwidth based on ``average'' requirements of the task; but this choice would result into transient degradations of the provided QoS that might be annoying. On the other hand, a bandwidth allocation based on worst case assumptions would most times be inefficient in terms of CPU utilisation. Moreover, usually is not easy to estimate the bandwidth needed as in the case of MPEG decoder. This problem can be addressed by dynamically adapting the amount of resources reserved to each task (i.e. by using a feedback inside the scheduling mechanism).

Feedback control techniques have been recently applied to real-time scheduling [Ek99] [Nak98][Reg01][Lip98][cer02][Stan02] and multimedia systems [Ste99]. Owing to the difficulties in modeling schedulers as dynamic systems, these works only provide a limited mathematical analysis of the closed-loop performance, often based on approximate models or intuitive arguments. The application of feedback to RB algorithms was pioneered in [Abe99-3] introducing the concept of *adaptive reservations.* This work opened up a new research thread. In [Abe02], it is shown how it is possible to write an exact mathematical model for the dynamic evolution of a single reservation and to design a switching Proportional Integer (PI) controller based on a linearisation of the system. Stability results and synthesis techniques for tuning the parameters of the switching PI controller, based on the theory of hybrid systems and on convex optimisation were shown in [Pal03].

The problem was further investigated in [Pal03-2], where a nonlinear feedback control scheme taking advantage of the specific structure of the system model was shown.

3.2 Novel contributions

We present here two novel contributions with respect to our previous work. First, we introduce novel control techniques, which have been designed by attacking the problem in the domain of stochastic control and stochastic dynamic programming.

In particular, we advocate a scheme where a dedicated controller is attached to each task. At each step the controller tries to optimise or decide the expected values of certain quantities of interest based on the expected behaviour of the computation times stochastic process.

To this end, we propose an architecture in which a separate component, the *predictor*, is responsible for providing the necessary information, based on its knowledge of the past evolution of the system.

Then, we propose a software architecture for feedback control. The architecture has been implemented in OCERA as part of this workpackage. Taking advantage of the Linux dynamic loadable module mechanism, the structure of our architecture is layered and

modular in its turn. The resource reservation scheduler is available as a separate component, while different control modules can be plugged in and out at the user's convenience.

3.3 System model

We consider a set of independent tasks $T^{(1)}$, ..., $T^{(n)}$ sharing a CPU. A task $T^{(i)}$ consists of a stream of jobs, or instances, $J_k^{(i)}$. Each job $J_k^{(i)}$ arrives (becomes executable) at time $r_k^{(i)}$, and finishes at time $f_k^{(i)}$ after executing for a time $c_k^{(i)}$. Job $J_k^{(i)}$ is associated a deadline $d_k^{(i)}$, which is respected if $f_k^{(i)} \leqslant d_k^{(i)}$, and is missed if $f_k^{(i)} > d_k^{(i)}$.

For our purposes, the sequences of computation times $\{c_k^{(i)}\}_{k \in N}$ are considered as discrete-time continuous valued stochastic processes.

For the sake of simplicity, we will restrict to *periodic* tasks}, in which $r_{k+1}^{(i)} = r_k^{(i)} + T^{(i)}$, where $T^{(i)}$ is the *task period*. Moreover, we will assume that $d_{k+1}^{(i)} = d_k^{(i)} + T^{(i)}$; hence, $r_{k+1}^{(i)} = d_k^{(i)}$.

For scheduling such tasks, we will use a resource reservation scheduler, namely the IRIS scheduler presented in the previous chapter. A very important property ensured by resource reservation scheduling is the so called *temporal isolation*, i.e. a task's schedulability depends only on the behaviour of the task itself and on the assigned budget $Q^{(i)}$. Thanks to this property, the task can be thought of as running on a *virtual CPU* having speed a fraction $B^{(i)}$ of the CPU speed. In fact, defining the *virtual finishing time* $v_k^{(i)}$ as the time the k^{th} job would finish if it were running on a virtual CPU with speed $B^{(i)}$, the enforcement of a hard reservation policy implies the following relation [Lip00]:

$$v_k^{(i)} - \delta \leqslant f_k^{(i)} \leqslant v_k^{(i)} + \delta \qquad (1)$$

where $\delta = (1 - B^{(i)}) P^{(i)}$. The above shows that in principle a resource reservation scheduler can be made to approximate a ``fluid" allocation of the processor as closely as needed by choosing $P^{(i)}$ small enough. However, in practical implementations, the overhead of context switches becomes relevant if $P^{(i)}$ is too small.

A consistency relation necessary for a resource reservation scheduler to work properly is

$$\sum_i B^{(i)} \leqslant U^{lub} \qquad (2)$$

with $U^{lub} \leqslant 1$ depending on the algorithm used for the implementation.

Model of the scheduler

When considering soft real-time applications it is of paramount importance to quantify the Quality of Service that each task experiences during his execution. In our model we can tolerate occasional deadline misses as long as the anomaly is kept in check.

Therefore, it is reasonable to define a quality of service metric, that we will call *scheduling error*, related to the deviation of the finishing time from the deadline. A possible definition for such a metric could be $e_k^{(i)}=(f_{k-1}^{(i)}-d_{k-1}^{(i)})/T_i$, where $e_k^{(i)}$ is the scheduling error experienced by job $J_{k-1}^{(i)}$. An ideal bandwidth allocation would be one for which $e_k^{(i)}=0$ for all k. Indeed, both $e_k^{(i)}>0$ and $e_k^{(i)}<0$ are undesirable situations, since in the former the task does not respect its timing constraint, whilst in the latter it receives an excess of bandwidth that would better be allocated to other activities.

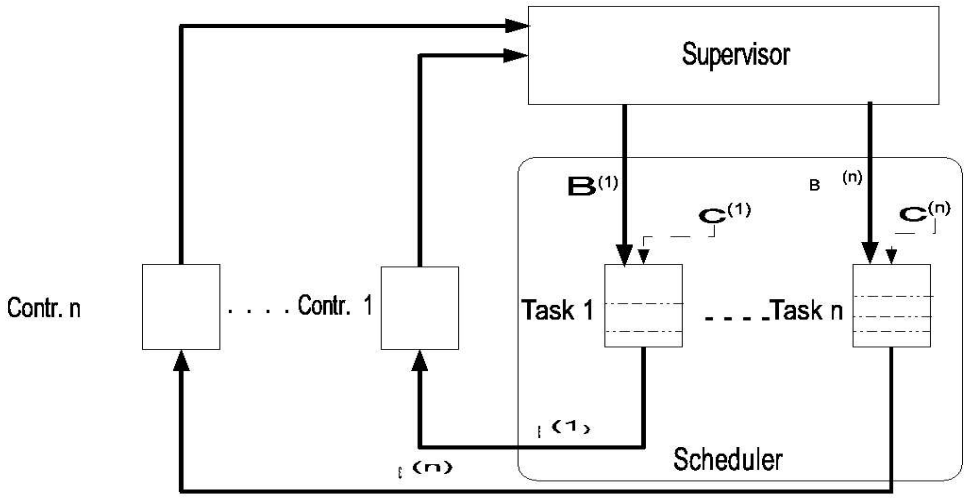
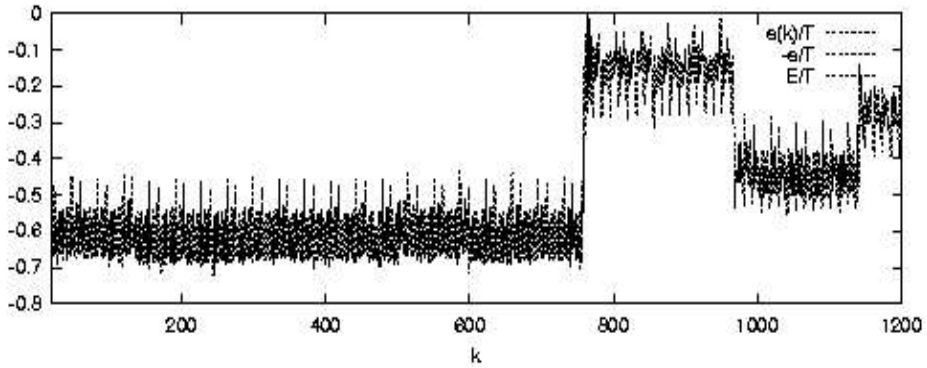


Figure 3.1Pictorial representation of the envisioned architecture: each task is controlled by a dedicate controller while a supervisor enforces the utilization bound condition.

(b)



(c)

Figure 3.2 Scheduling error for a static bandwidth scheduling of an MPEGplayer.

The introduction of a QoS metric exposes the limitations of resource reservation scheduling *per se*. Consider Figure 3.2, where we show the evolution of the scheduling

error for a multimedia task (MPEG decoding). Figure 3.2(a) reports the sequence of computation times for decoding a fragment of a Rock Concert movie (courtesy of Philips Research). The processor used for decoding is a Philips Nexperia trimedia and the framerate is 25 frame/sec. Computation times fluctuate around a mean value that is subject to sudden changes over time, due to the transitions from slow-moving scenes to quicker ones, and vice versa. The two bottom rows report simulation data for a static assignment of bandwidth. In the first experiment we chose a bandwidth equal to 1.3 times the mean of computation times divided by the task's period. The resulting scheduling error is shown in Figure 3.2(b): while the average computed over the sequence is acceptable there are long intervals of time when the scheduling error is large thus degrading unacceptably the experienced Quality of Service. Figure 3.2 (c), instead, shows what happens if the allocated bandwidth is calibrated on the worst case execution time. The scheduling error is always negative, but it has a large absolute value, so it results in a constantly large jitter value, meaning that the allocated bandwidth for the task is most times in excess.

The considerations above clearly motivate the need for a dynamic adaptation of the bandwidth a task is allocated during its execution, thus the idea of *adaptive reservation*. In particular, in the line of research initiated in [Abe99-3], we perform bandwidth adaptation using conceptual tools borrowed from feedback control theory. This concept is henceforth referred to as *feedback scheduling*.

Dynamic model

In order to design a feedback control we need a mathematical model for the system dynamic evolution. To this regard, the scheduling error as defined above, although an appealing QoS metric, turns out to be cumbersome to use. Instead, we shall define a different metric, by approximating the actual finishing time f_k of each job with its *virtual* finishing time, v_k :

$\epsilon_k^{(i)} = \frac{v_k^{(i)} - d_k^{(i)}}{T^{(i)}}$. In view of (1), it is easy to show that $\epsilon_k^{(i)}$ constitutes an approximation of the original metric $e_k^{(i)}$:

$$\epsilon_k^{(i)} - \delta' \leqslant e_k^{(i)} \leqslant \epsilon_k^{(i)} + \delta' \qquad (3)$$

(where $\delta' = \frac{\delta}{T} = (1 - B^{(i)}) \frac{P^{(i)}}{T^{(i)}}$), which clearly shows that the introduced approximation is acceptable provided that the ratio $\frac{P^{(i)}}{T^{(i)}}$ be small enough. The dynamics of $\epsilon_k^{(i)}$ is given by [Abe02]:

$$\epsilon_{k+1}^{(i)} = S(\epsilon_k^{(i)}) + \frac{c_k^{(i)}}{T^{(i)} B_k^{(i)}} - 1 \qquad (4)$$

where $S(x) = 0$ if $x < 0$ and $S(x) = x$ if $x \leqslant 0$.

For most resource reservation algorithms, $\epsilon_k^{(i)}$ is exactly and easily measurable upon the termination of each job.

Control goal

The above introduced concepts on resource reservation scheduling, and the model for the task evolution, allow a formulation of the control goal. As we said earlier, ideally one would wish to have the scheduling error always equal to zero. According to Equation (4), this would entail choosing $B_k^{(i)} = c^{(i)} / T^{(i)}$, which is evidently impossible without a prior knowledge of $\{c_k^{(i)}\}$. As a matter of fact, $\{\epsilon_k^{(i)}\}_{k \in N}$ are stochastic processes and reasonable design goals for the QoS can be formulated on:

- the first order probability density distribution $f_{\epsilon_k^{(i)}}(.)$: it can be used to make a qualitative comparison of two different control algorithms, by plotting the resulting distributions on the same graph;
- the expected value of the s.e. $\mu_{\epsilon_k^{(i)}} = E\{\epsilon_k^{(i)}\}$ and its variance $\sigma^2_{\epsilon_k^{(i)}} = E\{(\epsilon_k^{(i)} - \mu_{\epsilon_k^{(i)}})^2\}$:these values can be used for a quantitative comparison of two control techniques;
- the probability for the scheduling error $\epsilon_k^{(i)}$ to fall in a specified segment $[-e^{(i)}, E^{(i)}]$ of the real axis.

3.4 The feedback controller

Equation (4) describes a first order switching system, in which $\epsilon_k^{(i)}$ is a measurable state variable that we want to control, the bandwidth $b^{(i)}$ acts as a command variable, whereas $c_k^{(i)}$ is an exogenous disturbance term.

As a matter of fact, we have a collection of first order systems that evolve asynchronously one another, their states being observed at asynchronous points in time (jobs termination for the different tasks).

The asynchronicity of the system makes it difficult to design a global controller. A simpler choice is a decentralised scheme where a dedicated controller decides the bandwidth of each task looking at the evolution of the task itself in isolation. This idea is not completely applicable since the bandwidths chosen by the different controllers undergo a global constraint dictated by Equation (2). A minor departure from the entirely decentralised scheme is to include a supervisor that, whenever the controllers violate the constraint, resets the values of the bandwidths to fix the problem (e.g. operating a weighted compression or a saturation). From the standpoint of each controller, every time the supervisor is forced to act an impulsive disturbance is experienced (see Figure 3.1). This functionality is located in the QOS Supervisor module in Figure 1.1.

Single controller general design

The control scheme just introduced consists a collection of controllers attached to each task and a supervisor that performs corrective actions only when a controller chooses a value for the bandwidth in contrast with Equation (2) determining an overload condition. The latter component is described in depth in [Abe02-Th] and we will omit further details. Rather, this section is mainly concerned with the design of the dedicated controllers. In order to reduce the probability of overload conditions, and the subsequent

supervisory corrections, each controller is constrained by a ``local" saturation constraint: $b_k^{(i)} \leqslant B_{max}^{(i)}$.

Even choosing the saturation values so that $\sum_i B_{max}^{(i)} \geqslant U_{lub}$, their presence allows one to pose an upper bound on intensity of the disturbance term that can occur in presence of a supervisor correction.

From now on, we will concentrate on how to design a controller for a single task and the (i) superscript will be dropped for notational convenience. Clearly, the control problem would be trivial if the computation time c_k were known before beginning the k^{th} job.

To compensate for the lack of this knowledge, we propose a scheme based on two components (see Figure 3.1) : 1) a predictor, upon the termination of J_{k-1} , supplies a set of parameters I_k related to a prediction of c_k ; 2) a controller that decides the bandwidth b_k based on the set of parameters I_k and on the measurements of ϵ_k collected from the scheduler. The predictor plays in this scheme an important role: the more accurate the prediction the better the resulting control performance.

The ability to build an accurate predictor is related to the stochastic properties of the input process. A very simple predictor is one which is based on statistics (e.g. moving average) gathered on the past computation times. Actually, we will show that the type of information that the predictor needs to supply depends on the control scheme.

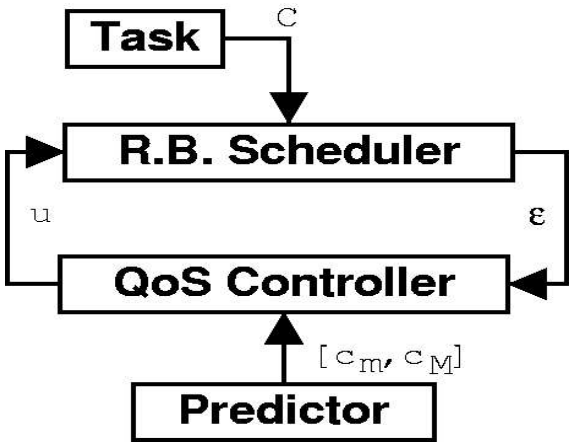


Figure 3.3Block diagram for QoS controller

In the rest of the section we shall show three different control techniques:

1. invariant based control
2. stochastic dead beat control
3. cost optimal control

In this context, we will simply show the basic ideas and the structureof the controllers.

Formal proofs on the closed loop stability and other properties can be found in [Pal03-2] [Pal03b].

Following we present three feedback scheduling algorithms, then we will give some implementation details.

Invariant based design

This control scheme has already been presented in [Pal03-2]. We report its description here for the sake of completeness and to compare its performance to other control schemes. The goal of an invariant based controller is to constrain the scheduling error evolution within a small region $[-e, E]$, compensating for the fluctuations of c_k. The information I_k provided at each step by the predictor is in this case a range $[h_k, H_k]$ where the next computation time c_k is expected to fall. Assuming that $c_k \in [h_k, H_k]$ (correct prediction) the controller is required to behave as follows:

- if ϵ_k belongs to the set $[-e, E]$ also ϵ_{k+1} has to belong to the same set (invariance mode)
- if ϵ_k is outside of $[-e, E]$ it will be steered back into $[-e, E]$ in a predetermined number of steps (recovery mode)

Whenever the computation time deviates from the predicted range, it is possible that the scheduling error exits the invariant region, thus the *recovery* control mode is used to steer it back into the region.

A theoretical discussion on conditions for such a controller to exist as well as on the problem of mistaken predictions (i.e. $c_k \notin [h_k, H_k]$) can be found in the cited paper. In this context we just summarise results on how to choose the bandwidth:

[step k)] choose b_k

$$\begin{aligned} &\in \left[\frac{H_k}{T(1+E-S(\epsilon_k))}, \frac{h_k}{T(1-e-S(\epsilon_k))} \right] \ \text{if } \epsilon_k \leqslant \epsilon^1 \\ &\in \left[\frac{H_k}{T(1+E-S(\epsilon_k))}, B_{max} \right] \ \text{if } \epsilon^1 < \epsilon_k \leqslant \epsilon^2 \\ &= B_{max} \ \text{if } \epsilon_k > \epsilon^2 \end{aligned} \tag{5}$$

where $\epsilon^1 = 1 - e - \frac{h_k}{T B_{max}}$ and $\epsilon^2 = 1 + E - \frac{H_k}{T B_{max}}$.

[step 0)] choose b_0 in the same range as for a negative scheduling error.

The control formula just showed embeds the simplest recovery policy, which assigns the maximum available bandwidth in such situations. Though, alternative policies are also possible, aiming at achieving a proper trade-off between the speed of the recovery and the expense in terms of used bandwidth. For example, it is possible to force an exponential reduction of the gap between the scheduling error value and the invariance region. This is discussed further in [Pal03b].

Stochastic dead beat approach

This control scheme attacks the design problem in the stochastic domain. The goal is to choose a bandwidth such that the expectation of the next scheduling error be equal to a desired value. The expectation that we are considering is conditioned to the past evolution of the system. If the desired value is zero we refer to the controller as Stochastic Dead Beat (SDB). It is possible to prove that the control law having such a property, and satisfying the saturation constraint, can be expressed as follows:

$$b_k = \begin{cases} \dfrac{\mu_{C_k}}{T(1-s(\epsilon_k))} & if\ \epsilon_k < 1 - \dfrac{\mu_{C_k}}{B_{max}} \\ B_{max} & if\ \epsilon_k \geqslant 1 - \dfrac{\mu_{C_k}}{B_{max}} \end{cases} \quad (6)$$

If $\epsilon_k > 1 - \dfrac{\mu_{C_k}}{TB_{max}}$, then it is not possible to guarantee that the expected next error be zero. For this control scheme the information I_k required from the predictor is μ_{C_k} , i.e. the expectation of c_k conditioned to the past evolution of the system. This can be done, for example, with a moving average performed on last execution times. Despite its simplicity, this technique is able to achieve a very good performance, as we will show in Section~\ref{sec:experiments}.

Optimal cost approaches

This technique is also based on the framework of stochastic control. In particular, taking inspiration from dynamic programming techniques [Ros83], the controller chooses the value for the bandwidth B_k so as to optimise the expectation $\overline{w}(\epsilon, b)$ (conditioned to the past evolution of the system) of a cost function \$ $w(\epsilon, ., b)$. Such a function expresses, at step k , the cost to pay if we choose the bandwidth value $b_k = b$, if the achieved next system state is $\epsilon_{k+1} = \epsilon$.

In particular we chose a cost function accounting for the deviation of the next scheduling error from zero, and the bandwidth being used:

$w(\epsilon_{k+1}, b) = \gamma \epsilon_{k+1}^2 + (1-\gamma) b$, where $\gamma \in (0,1)$ allows us to assign different weights to the scheduling error or to the used bandwidth.

In case $\epsilon_k = 1$, the minimum is immediately found as $b_k = \sqrt{[3]}2 \dfrac{\gamma}{1-\gamma}(\sigma^2 + \mu^2)$

In the other cases the following formula holds:

$$
\begin{aligned}
b_k(\epsilon_k) &= sqrt[3]\rho + \delta(\epsilon_k) + \sqrt{[3]}\rho - \delta(\epsilon_k) \\
\rho &= \frac{\gamma(\sigma^2+\mu^2)}{(1-\gamma)} \\
\delta(\epsilon) &= \sqrt{\left(\frac{\gamma}{1-\gamma}\right)^2\left(\sigma^2+\mu^2\right)^2 + \left(\frac{2}{3}\frac{\mu\,\gamma\,[1-S(\epsilon_k)]}{1-\gamma}\right)^3}
\end{aligned}
$$

This formula can be directly used for all $\epsilon_k \leqslant \epsilon' = 1 + \frac{3}{2\mu_C}\sqrt{[3]}\frac{\gamma}{1-\gamma}(\sigma^2+\mu^2)$,

which is the range for which $\delta(\epsilon_k)$ is real. For $\epsilon_k > \epsilon'$, the formula still holds if computations are properly performed in the complex domain. Furthermore, note that the optimum bandwidth value found with this formula is subject to the usual saturation constraint due to B_{max} . As for the case of SDB, we used μ for the expectation of c_k , while σ denotes its standard deviation. Both quantities are conditioned to the past evolution of the system and are the required output of the predictor for this control scheme.

Figure 3.4 reports the optimal $B(\epsilon_k)$ function for a particular set of parameters. The same figure makes also a comparison with the bandwidth function in Equation (6).

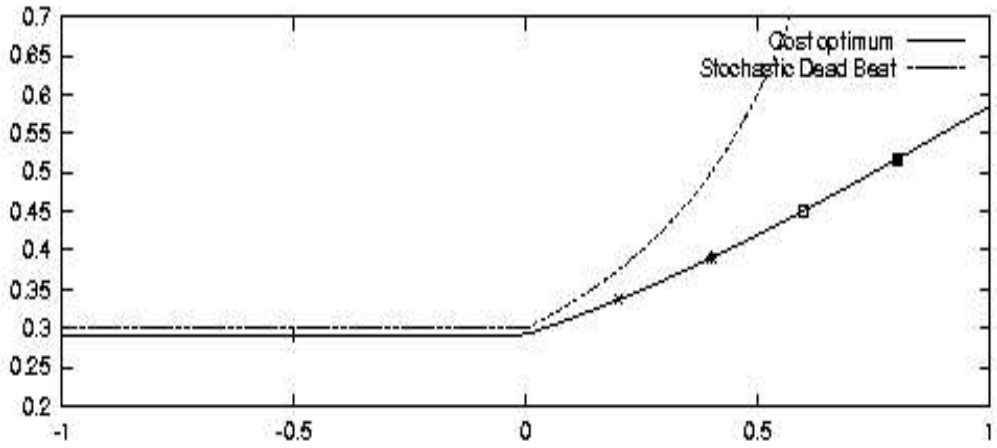


Figure 3.4 Optimal B(.) function for the optimal cost approach compared with SDB.

An important problem with this approach is that the computation of the bandwidth requires several floating point operations for which it is not immediate to achieve an efficient kernel implementation. For fixed μ and σ the problem is relatively simpler in that it is possible to do efficient linear interpolations of the curve. For dynamically changing parameters, more sophisticated techniques are required and they are currently under investigation.

Minimum expected square scheduling error

A special case for the technique shown above is when $\gamma = 1$. In this case, the controller minimises, at each step, the expectation of the squared value of the next scheduling error, subject to the saturation constraint. The optimisation problem yields, in this case, a simpler formula: control law:

$$B(\epsilon_k) = \begin{cases} \dfrac{\sigma^2 + \mu^2}{\mu\,[1 - S(\epsilon)]} & if\ , \epsilon < 1 - \dfrac{\sigma^2 + \mu^2}{\mu\ B_{max}} \\ B_{max} & if\ \epsilon_k \geqslant 1 - \dfrac{\sigma^2 + \mu^2}{\mu\ B_{max}} \end{cases} \qquad (7)$$

Note that this solution is only valid if $B_{max} > \dfrac{\sigma^2 + \mu^2}{\mu} = \mu + \dfrac{\sigma^2}{\mu}$.

If such relation does not hold, the optimal control reduces to the trivial law always returning B_{max} .

The optimal bandwidth assignment that we got is is equal to the one given by the SDB formula (6), plus a factor that is proportional to the input process variance σ^2 . Perfect equivalence with SDB is there only in the limit for $\sigma \to 0$. This is reasonable: since here we want to minimise the squared value of ϵ_k rather than its expectation, we have to take into account the standard deviation σ using larger bandwidth valued to compensate for it.

3.5 Implementation of the QoS manager

In general, different applications may need different controller strategies. Each module will manage all tasks with the same characteristics, that need to be served by the same control algorithm. A task can choose the QoS manager for its execution by specifying, in the sched_setscheduler() call, the SCHED_QMGR1, SCHED_QMGR2, etc ... scheduling policy, and by providing proper parameters to the module through the sched_param structure.

```
int main()
{
    // initialization
    sched_param param;
    // init controller parameters
    sched_setscheduler(mypid, SCHED_QMGR1, &param);
    while (1) {
      //main loop code
      qmgr_end_cycle();
      qmgr_wait_period();
    }
}
```

Figure 3.5 Typical structure of a cyclic task.

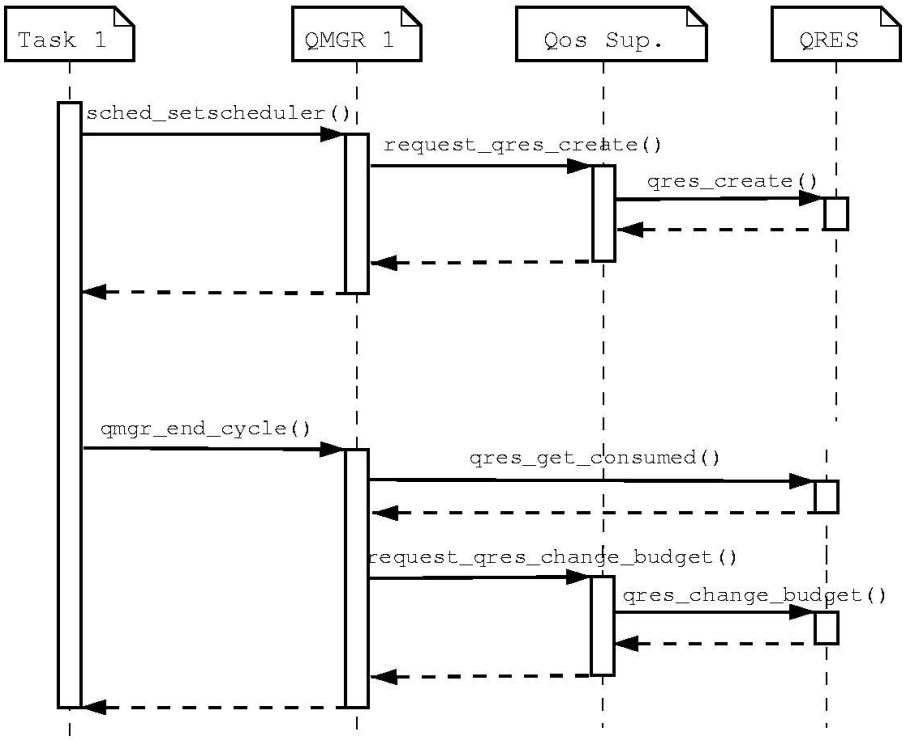


Figure 3.6Sequence diagram that shows the interaction between the QMGR the QSPV and the QRES modules.

A task using adaptive reservations must be linked against the qos library, that provides some commodity function implementing the user-level part of the feedback strategy. Recall that only periodic tasks are considered. As a result, a task attached to an adaptive reservation will have the structure shown in Figure 3.5 (the sequence of invocations is shown in Figure 3.6 as a sequence diagram).

At the beginning, the task must perform an initialization phase in which the setsched_hook of the QMGR1 module is invoked. After storing the controller parameters in its internal data structures, the QMGR1 module invokes the qspv_request _create() function of the QSPV module to initialize the reservation budget and period.

After initialization, the task enters a loop. Each execution of the loop corresponds to a job of the task. For example, in case of a MPEG decoder, a job may correspond to the decoding of one frame. At the end of the loop, the task signals the QMGR the termination of the job by invoking the qmgr_end_cycle() function provided by the qoslib.

The qoslib will then call the proper QMGR1 handler, which, in turn, calls the qres_get_consumed() function of the QRES module to obtain the amount of budget consumed by the job.

Then, the control law is applied and a new budget is computed and set with the qspv_change _budget() function of the QSPV module. If there is not enough free bandwidth to accommodate for the new budget, the QoS supervisor can implement three possible behaviours, similar to the ones described in the previous section: *saturation*, *compression*, or *reject*. In case the saturation policy is selected, the highest possible budget is assigned to the task so that the total CPU utilization does not exceed U_{lub} .

In case the compression policy is selected, all the reservations are recomputed (``compressed'') so that we can make enough space for the new request (see [Abe99-3], [Abe02-Th]). In case the reject policy is specified, the bandwidth adaptation fails (i.e., the budget is not changed). In any case, the qspv_change_budget() returns the actual value of the budget that has been set.

Finally, the task blocks waiting for the next periodic event by calling the wait_period()} function provided by qoslib. The periodic behaviour of the task is application dependent. In other words, it is the responsibility of the application to set up a periodic timer event and to block waiting for the event (although the qoslib provides some helper functions for setting up periodic tasks).

3.6 Experimental results

In this section we report experimental results gathered on a real Linux system. The considered application is a MPEG decoder. While the OS infrastructure described above is at advanced testing stage, the adaptation of a MPEG player (namely, the *xine* player) is still under way. Therefore, we emulated the behaviour of the decoder by a task that periodically reads a trace file and, for each job, consumes a time equal to the one read from the file (by a time consuming loop). The trace file has been provided by Philips Research labs and refers to the same movie the segment in next figure is taken from.

In the first set of experiment, we wanted to gauge the benefit of the feedback scheduling mechanism. In the second set of experiments we compared the performance of different controllers. Finally, in the third set of experiments, we evaluated the influence of the predictor component.

Benefits of feedback

Consider again the MPEG decoding times shown in Figure 3.2. Figure 3.7 shows the scheduling error evolution that is achieved when the bandwidth is allocated by an invariant-based QoS controller, for the same input sequence. The predictor, in this case, produces at each step an interval $[h_k, H_k]$ based on moving averages of the last ten samples. In particular $h_k = \mu_k - \sigma_k$ $ and $H_k = \mu_k + \sigma_k$ where μ_k is the moving average and σ_k is the moving variance. The invariant region $[-e, E]$ was fixed apriori and the controller law is chosen in mid-point of the admissible range (Equation (5)).

The only significant deviation from the target set (around the 790th sample) is due to a swift scene change, which messes up the moving average predictor, but is a transient problem soon recovered. A visual comparison between Figures 3.2 and 3.7 is illustrative of the extent of the achieved performance enhancement.

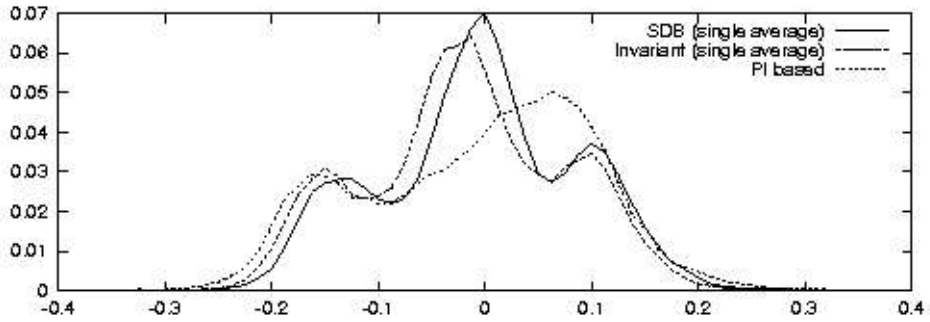


Figure 3.8Comparison of the performance achieved by traditional PI-based, invariant-based and stochastic dead beat controllers, while playing an MPEG movie

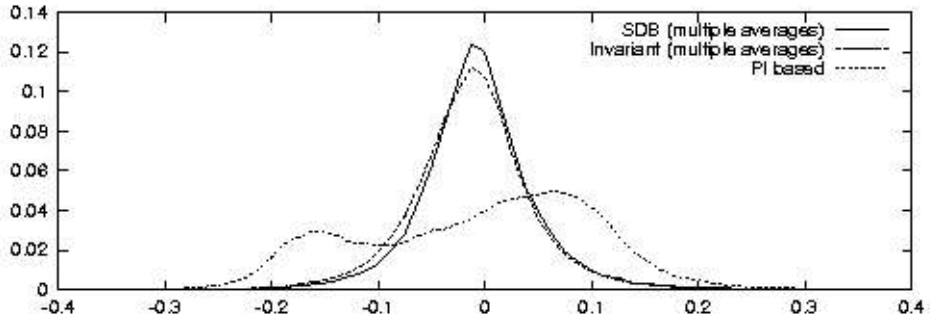


Figure 3.9Comparison of the performance achieved by traditional PI-based, invariant-based and stochastic dead beat controllers, when multiple averages are used in last two types of control.

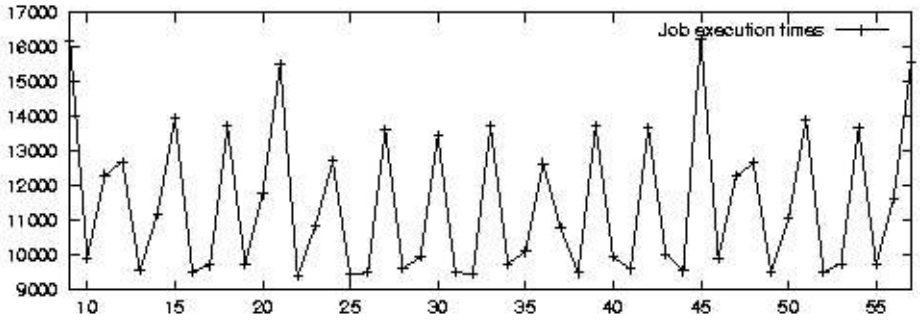


Figure 3.10 Job execution times for an MPEG Movie with fixed frame types pattern IBBPBBPBBPBB.

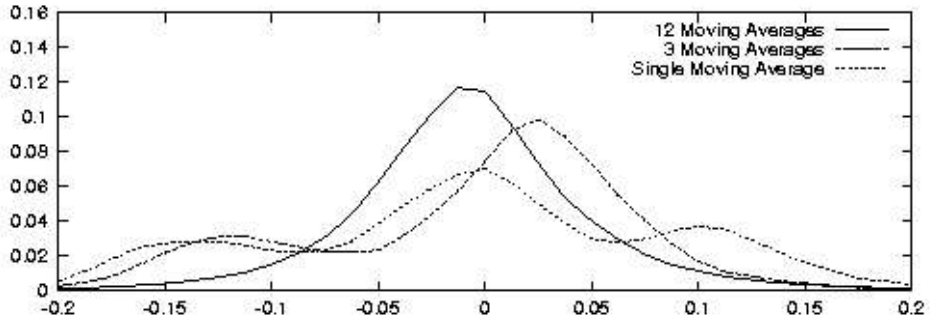


Figure 3.11Scheduling error PDF obtained with a single moving average vs. 3 and 12 multiple moving averages.

Comparing PI with invariant-based

For the purpose of evaluating the performance achieved by our approach, it is useful to make comparisons with alternative approaches. A feedback controller that has been studied in the past is a classic linear PI controller, like the one introduced in [PalAbe]. In that approach, the authors used a controller assigning the bandwidth at each step depending on the value of the scheduling error at the current and previous steps (linear action), and on the bandwidth value assigned at the previous step (integral action). Statistics have been gathered by a run of that controller, and a run of the invariant-based controller as introduced in this work, with exactly the same trace of decoding times from an MPEG2 movie. Obtained scheduling error PMFs are shown in Figure 3.8. The figure highlights that an invariant based control achieves a PMF very similar to the one achieved by a stochastic dead beat control. Both of them manage to keep the scheduling error within a region near the origin, in a tighter manner with respect to what has been achieved by using a PI based control. The parameters of the controllers have been chosen so to perform a comparison at similar achieved mean values for the scheduling error, and a single moving average has been used in order to estimate the input process statistics. Clearly, by using multiple moving averages, the approaches introduced in this paper achieve a much tighter scheduling error distribution, around the origin, as shown in Figure 3.9.

Predicting computation times

The approach to QoS control in scheduling of soft real-time tasks proposed so far is based on the knowledge of a quite small interval $[c_m(k), c_M(k)]$ in which the next job execution time is supposed to reside with a high probability (in the probabilistic bound model). The way such a prediction can be performed is highly application-dependent. In this paragraph, a brief example shows how the proper choice of a predictor for job execution times can dramatically improve performance of the QoS controller relative to an MPEG decoder. A common class of MPEG movies has a periodic structure in the frame types, i.e. there exist a fixed sequence of frame types that repeats over and over during the movie (a common example is IBBPBBPBBPBBIBBP...). Decoding times for various frame types are typically different, so that, looking closely at the job execution times, it is possible to notice a periodic structure repeating all over the movie (see Fig. 3.10). In such a case, a simple moving average (plus a moving standard deviation) among job execution times completely fails in helping predicting the position of the next sample, because the periodic load peaks corresponding to frames of type 'I' cannot be predicted. Instead, it is much better to consider a different moving average for each different class of load level. Figure 3.11 highlights the performance improvement achieved with an approach of this kind. In fact, the scheduling error PDF is much more narrow, resulting in a much more effective task control. The figure shows results obtained with a variable number of moving averages, each one operating every 3 and 12 execution time samples.

Bibliography

Abe02: Luca Abeni and Luigi Palopoli and Giuseppe Lipari and Jonathan Walpol, Analysis of a Reservation-Based Feedback Scheduler,Proc. of the Real-Time Systems Symposium, 2002

Abe02-Th: Luca Abeni, Supporting time-sensitive Activities in a Desktop Environment,, 2002

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

Abe99: Luca Abeni and Giorgio Buttazzo, Constant Bandwidth vs. Proportional Share Resource Allocation,Proceedings of the IEEE International Conference on Mutimedia Computing and Systems, 1999

Abe99-3: Luca Abeni and Giorgio Buttazzo, Adaptive Bandwidth Reservation for Multimedia Computing,Proceedings of the IEEE Real Time Computing System, 1999

cer02: Cervin, Anton and Eker, Johan and Bernhardsson, Bo and AA'en, Karl-Erik, Feedback-Feedforward Scheduling of Control Tasks, , 2002

Ek99: Johan Eker, Flexible Embedded Control Systems: Design and Implementation,, 1999

Lip00: G.Lipari and S.K. Baruah, Greedy reclaimation of unused bandwidth in constant bandwidth servers,IEEE Proceedings of the 12th Euromicro Conference, 2000

Lip98: GiuSeptemberpe Lipari and Giorgio Buttazzo and Luca Abeni, A Bandwidth Reservation Algorithm for Multi-Application Systems,IEEE Real Time Computing Systems and Applications, 1998

mar04: L.Marzario, G. Lipari, P. Balbastre, A. Crespo, IRIS: A new reclaiming algorithm for server-based real-time systems,10th IEEE Real-Time and Embedded Technology and Ap, 2004

Nak98: Tatsuo Nakajima, Resource Reservation for Adaptive QOS Mapping in Real-Time Mach,Sixth International Workshop on Parallel and Distr, 1998

Pal03: L. Palopoli and L. Abeni and G. Lipari, On the application of hybrid control to CPU Reservations,Hybrid systems Computation and Control (HSCC03), 2003

Pal03-2: Luigi Palopoli and Tommaso Cucinotta and Antonio Bicchi, Quality of service control in soft real-time applications,Proc. of the IEEE 2003 conference on Vision, December 2003

Pal03b: Luigi Palopoli and Tommaso Cucinotta, QoS control in reservation-based scheduling,, 2003

Raj97: R. Rajkumar, K. Juvva, A. Molano and S. Oikawa, Resource Kernels: a resource centric approach to real-time and multimedia,Real-Time Computing Systems and Applications, 1997

Reg01: John Regehr and John A. Stankovic, Augmented CPU Reservations: Towards Predictable Execution on General-Pur,Proceedings of the IEEE Real-Time Technology and A, 2001

Ros83: S.M. Ross, Introduction to stochastic dynamic programming,, 1983

Stan02: C. Lu, J. Stankovic, G. Tao and S. Son, Feedback Control Real-Time Scheduling: Framework, Modeling and Algorithms, , 2002

Ste99: David Steere and Ashvin Goel and Joshua Gruenberg and Dylan McNamee and Cal, A Feedback-driven Proportion Allocator for Real-Rate Scheduling,Proceedings of the Third usenix-osdi, 1999