QoS Management through adaptive reservations *

L. Abeni, T. Cucinotta, G. Lipari, L. Marzario, L. Palopoli
ReTiS Lab, Scuola Sup. Sant'Anna, Pisa, Italy

Abstract

*Reservation based (RB) scheduling is a class of scheduling algorithms that is well-suited for a large class of soft real-time applications. They are based on a "bandwidth" abstraction, meaning that a task is given the illusion of executing on a dedicated slower processor. In this context, a crucial design issue is deciding the bandwidth that each task should receive. The point we advocate is that, in presence of large fluctuations on the computation requirements of the tasks, it can be a beneficial choice to dynamically adapt the bandwidth based on QoS measurements and on the subsequent application of feedback control (*adaptive reservations*).*

In this paper, we present two novel contributions to this research area. First, we propose three new control algorithms inspired to the ideas of stochastic control. Second, we present a flexible and modular software architecture for adaptive reservations*. An important feature of this architecture is that it is realised by means of a minimally invasive set of modifications to the Linux kernel.*

1. Introduction

Software based implementation of time-sensitive applications is gaining momentum because it is generally regarded as cheaper and more flexible than a dedicated hardware solution. Important examples can be found in the area of consumer electronics: multimedia streaming programs, video/audio players, software sound mixers, movie editing, and so on. Another relevant area of application is offered by embedded systems used in data-intensive processing of sensor data, where high volumes of data have to be processed in real-time (i.e. radar systems). Such applications are characterised by implicit timing constraints (deadlines) for which occasional failures can be tolerable, provided that they do not become too frequent. For instance, when streaming a MPEG movie, the delayed decoding of a few frames is not even perceived by the user as long as the system behaves "well" in the average.

It often occurs that multiple applications (implemented by software tasks) populate the same system and compete for a pool of shared resources. In this case, an appropriate real-time scheduling solution has to be utilised to attain both an efficient use of the processor and an acceptable level of Quality of Service (QoS) for the different tasks. To this regard, traditional real-time scheduling techniques, such as Rate Monotonic(RM) and Earliest Deadline First (EDF) [16], have evident and well-known shortcomings. Indeed, because classical real-time scheduling theory deems unacceptable even a single deadline violation, schedulability tests are based on worst case assumptions for execution and inter-arrival time of tasks. However, the strict compliance with every deadline is irrelevant in this context and it leads to an overly conservative management of the CPU. On the other hand, it is crucial that large fluctuations on the execution or inter-arrival times of a task do not affect the performance levels granted to other tasks. This property is called *temporal isolation* (or temporal protection) and it is neither provided by RM, nor by EDF.

*This work has been partially supported by the European OCERA IST-2001-35102 and RECSYS IST-2001-32515 projects.

In the past years, several scheduling algorithms providing temporal isolation have been developed, ranging from Proportional Share (PS) [13] to Reservation Based (RB) [18] algorithms. Both these classes of algorithms can be shown to approximate a fluid allocation of the processor, giving each task the "illusion" of running on a dedicated slower processor. In particular, RB techniques approximate this behaviour by periodically granting a fixed amount of execution to each task.

RB techniques are the basis this work is developed upon. More specifically we consider CPU reservations, which have been implemented on a variety of systems using different algorithmic solutions [22, 3, 12, 23]. We are confident that most of the presented results can be extended to the management of other kind of resources, like network bandwidth and disk, and to algorithms providing temporal protection other than CPU reservations, such as the ones based on PS.

The traditional way for using RB 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 of the times be inefficient in terms of CPU utilisation. 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).

A first proposal for feedback based scheduling of time sharing systems dates back to 1962 [9]. More recently, feedback control techniques have been applied to real-time scheduling [11, 19, 24, 14, 17, 8, 7] and multimedia systems [29]. Owing to the difficulties in modelling 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 [4] introducing the concept of *adaptive reservations*. This work opened up a new research thread. In [6], 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 [20].

The problem was further investigated in [21], where a nonlinear feedback control scheme taking advantage of the specific structure of the system model was shown. In this paper, this control approach is shortly reviewed to the purpose of comparing it with novel feedback designs that we present here. An approach that bears some resemblance to the one presented in this paper is the one shown in [1].

Contributions of this paper In this paper, we present two novel contributions with respect to our previous work [20, 21]. 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 time 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. Throughout the paper we will shortly describe these control schemes and report a comparative evaluation of their performance with more traditional schemes.

The second important contribution of the paper is the description of a software architecture for our feedback control strategies. The problem has been discussed in the past few years and interesting proposals emerged at the middleware level [10, 31]. In this paper we take a different viewpoint, searching for an architectural support for our technology suitable for a general purpose operating system by a minimally invasive set of modifications. Namely, we show our implementation based on the Linux kernel [15]. Taking advantage of its modularity, the structure of

our architecture is layered and modular in its turn. The RB scheduler is available as a separate component, while different control modules can be plugged in and out at the user's convenience. Contrary to other approaches that are more focused on hard real-time applications, such as RTlinux [27] and RTAI [26], we can run time-sensitive applications in user space with obvious benefits in terms of safety and access to a wealth of libraries available for Linux. Modifications of the original kernel have been carried on in such a way that ordinary Linux applications can run without any awareness whatsoever of the new environment.

2. Problem presentation

2.1. The task model

We consider a set of independent tasks $\mathcal{T}^{(1)}, ..., \mathcal{T}^{(n)}$ sharing a CPU. A task $\mathcal{T}_i$ 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 $f_k^{(i)} \leq d_k^{(i)}$, and is missed if $f_k^{(i)} > d_k^{(i)}$.

For our purposes, the sequences of computation times $\{c_k^{(i)}\}_{k\in\mathbb{N}}$ 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, $r_{k+1}^{(i)} = d_k^{(i)}$.

2.2. Resource Based Scheduling

In the application that we will show in this paper, tasks are scheduled by a Reservation Based (RB) policy [18].

In a RB framework, a task $\mathcal{T}^{(i)}$ is associated a pair $(Q^{(i)}, P^{(i)})$, said *reservation*, meaning that the scheduling algorithm guarantees to $\mathcal{T}^{(i)}$ a *budget* of $Q^{(i)}$ execution time units in every *reservation period* $P^{(i)}$ (whenever in need). The ratio $B^{(i)} = Q^{(i)}/P^{(i)}$ is referred to as *bandwidth*. Dealing with periodic tasks, it is convenient to choose $P^{(i)}$ so that $T^{(i)} = kP^{(i)}, k \in \{1, 2, \ldots\}$. If the task is not allowed to execute for more than $Q^{(i)}$ units every $P^{(i)}$, even in presence of a idle processor, then the reservation is said *hard* [22]. In this paper we will restrict our attention to this class of RB algorithms, even though most techniques and considerations shown in the sequel are applicable to a good extent also to other types of reservations.

A very important property ensured by RB scheduling is the so called *temporal isolation*, which can be defined as follows.

Definition 1 *A scheduling algorithm is said to guarantee temporal protection if the ability for each task $\tau^{(i)}$ (with $i = 1, \ldots, n$) to meet its timing constraints depends only the evolution of the task's workload* [1].

Thanks to this property, the task can be thought of as running on a *virtual CPU* whose speed is a fraction $B^{(i)}$ of the CPU speed. In order to formally express this concept, define the *virtual finishing time* $\mathrm{v}_k^{(i)}$ as the time the k^{th} job would finish if it were running on a virtual CPU with speed $B^{(i)}$. The following statement proves that in principle a RB scheduler can be made to approximate a "fluid" allocation (see also [12]).

Fact 1 *Assume that a hard reservation policy is used to schedule task $\tau^{(i)}$ guaranteeing that it receives $Q^{(i)}$ computation units in ever server period $P^{(i)}$. The following relation holds true:*

$$\max\left\{\lfloor\frac{v_k^{(i)}}{P^{(i)}}\rfloor P^{(i)}, v_k^{(i)} - \delta\right\} \leq f_k^{(i)} \leq \min\left\{\lceil\frac{v_k^{(i)}}{P^{(i)}}\rceil P^{(i)}, v_k^{(i)} + \delta\right\}, \tag{1}$$

where $\delta = (1 - B^{(i)})P^{(i)}$.

[1] The workload of a periodic task is uniquely determined by its computation time and by its period

Proof:
A first consideration is that $v_k^{(i)}$ and $f_k^{(i)}$ lie in the same reservation period, i.e., $\lfloor \frac{v_k^{(i)}}{P^{(i)}} \rfloor P^{(i)} \leq f_k^{(i)} \leq \lceil \frac{v_k^{(i)}}{P^{(i)}} \rceil P^{(i)}$. Moreover, sharper bounds for their variation can be made considering the two following situations: 1) there remains an infinitesimal unit of computation to be carried out in the last reservation period, 2) there remains $Q^{(i)}$ units of computation to be carries out in the last period. Considering situation 1), the virtual finishing time is located right after the beginning of the last reservation period. In this case the worst case situation is when residual computation time is delayed to the maximum possible extent allowed by the reservation, i.e., $f^{(i)} \leq v^{(i)} + P^{(i)} - Q^{(i)}$. We can similarly deal with situation 2) leading to $f^{(i)} \geq v^{(i)} - P^{(i)} + Q^{(i)}$. Our claim is then proved. •

As a consequence of the above in principle a RB scheduler can be made to approximate a "fluid" allocation of the processor as closely as needed by choosing $P^{(i)}$ small enough, i.e., $\lim_{P^{(i)} \to 0} f_k^{(i)} = v_k^{(i)}$. However, in practical implementations, the overhead of context switches becomes relevant if $P^{(i)}$ is too small.

A consistency relation necessary for a RB scheduler to work properly is

$$\sum_i B^{(i)} \leq U^{lub}, \tag{2}$$

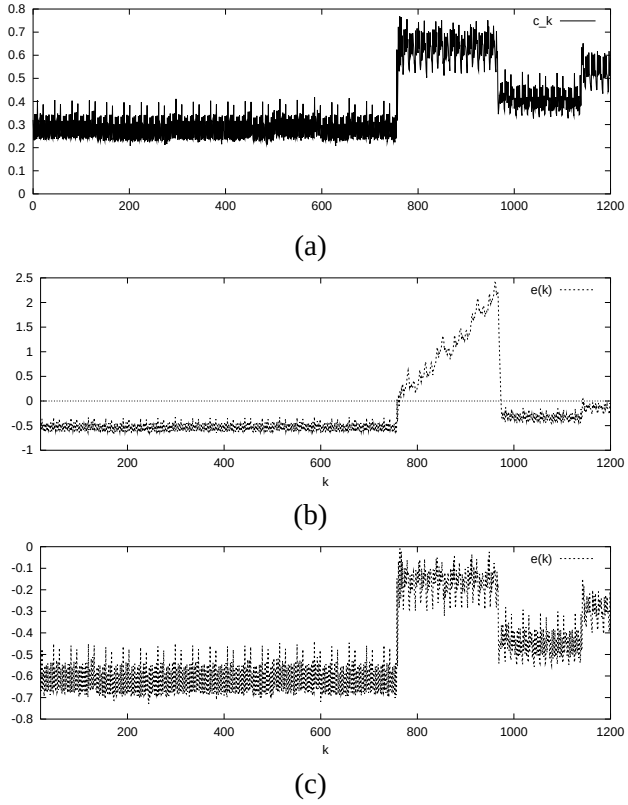
with $U^{lub} \leq 1$ depending on the algorithm used for the implementation.

2.3. Adaptive Reservations

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 case the task does not respect its timing constraint, while in the latter it receives an excess of bandwidth that would better be allocated to other activities.

The introduction of a QoS metric exposes the limitations of RB scheduling *per se*. Consider Figure 1, where we show the evolution of the scheduling error for a multimedia task (MPEG decoding). Figure 1(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 frame rate 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 1 (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 1 (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 [4], we perform bandwidth adaptation using conceptual tools borrowed from feedback control theory. This concept is henceforth referred to as *feedback scheduling*.



(a)

(b)

(c)

Figure 1. Scheduling error for a static bandwidth scheduling of an MPEG player.

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

$$\varepsilon_k^{(i)} - \delta^{'} \leq e_k^{(i)} \leq \varepsilon_k^{(i)} + \delta^{'}, \tag{3}$$

(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 $\varepsilon_k^{(i)}$ is given by [6]:

$$\varepsilon_{k+1}^{(i)} = S(\varepsilon_k^{(i)}) + \frac{c_k^{(i)}}{T^{(i)} B_k^{(i)}} - 1 \tag{4}$$

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

The bandwidth $B_k^{(i)}$ is in this case considered as a command variable ant it is thus allowed to vary over time. For most RB algorithms, $\varepsilon_k^{(i)}$ is exactly and easily measurable upon the termination of each job. For particular classes of RB, such as the Constant Bandwidth Server[3], different definitions for $\varepsilon_k^{(i)}$ are easier to deal with; for the sake of brevity we will not touch this issue (the interested reader is referred to [6]).

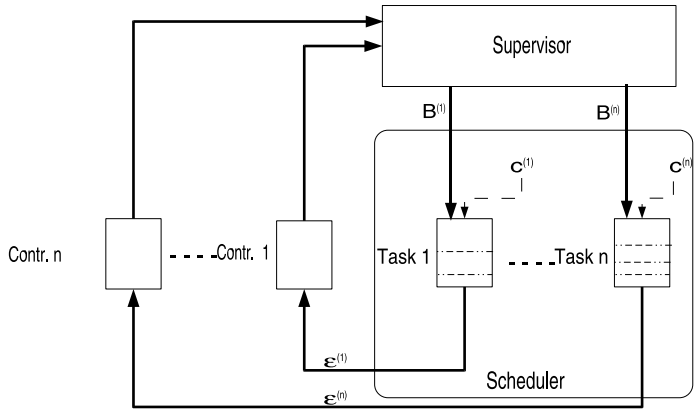


Figure 2. Pictorial representation of the envisioned architecture: each task is controlled by a dedicate controller while a supervisor enforces the consistency condition $\sum B^{(i)} \leq 1$.

2.5. Control performance

The above introduced concepts on RB scheduling, and the model for the task evolution, allow us to formally express the control goals and to gauge the quality of the attained result. As we said earlier, one would ideally wish to have the scheduling error always equal to zero. According to Equation (4), this would entail choosing $B_k^{(i)} = c_k^{(i)}/T^{(i)}$, which is evidently impossible without a prior knowledge of $\{c_k^{(i)}\}$. As a matter of fact, $\{\varepsilon_k^{(i)}\}_{k\in\mathbb{N}}$ are stochastic processes. Therefore a reasonable formulation on the desired behaviour necessarily regards their stochastic properties. A first possibility is on the "shape" of the first order density distribution $f_{\varepsilon_k^{(i)}}(\cdot)$. Ideally, we would wish it to be as close as possible to a Dirac delta centred on 0 (meaning that the scheduling error at step k is 0 with probability 1). Based on this expectation, we can make a qualitative comparison of different control algorithms evaluating how much they are close to this ideal performance. A more quantitative evaluation can be made considering the expected value of the squared scheduling error $E\left[(\varepsilon_k^{(i)})^2\right]$ that accounts for average deviation of this quantity in both directions (clearly the smaller this quantity, the better the performance). Finally a possible control performance specification (and a subsequent control scheme) can be made by requiring that the scheduling error $\varepsilon_k^{(i)}$ reside with a good probability in a specified segment $[-e^{(i)}, E^{(i)}]$.

3. Feedback scheduling techniques

Equation (4) describes a first order switching system, in which $\varepsilon_k^{(i)}$ 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 asynchronism 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

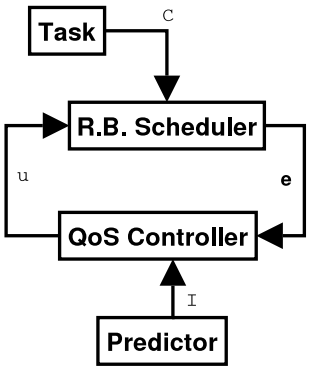


Figure 3. Block diagram for QoS controller

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 2).

3.1. Single controller general design

The control scheme just introduced consists of a collection of controllers attached to each task and of 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 [2] 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: $B_k^{(i)} \leq B_{max}^{(i)}$. Even choosing the saturation values so that $\sum_i B_{max}^{(i)} \geq U_{lub}$, 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) 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 an important role in this scheme: 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.

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 structure of the controllers. Formal proofs on the closed loop stability and other properties can be found in [21].

3.2. Invariant based design

This control scheme has already been presented in [21]. 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:

1. if ε_k belongs to the set $[-e, E]$ also ε_{k+1} has to belong to the same set (invariance mode)
2. 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 such that:

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

where $\varepsilon^1 = 1 - e - \frac{h_k}{TB_{max}}$ and $\varepsilon^2 = 1 + E - \frac{H_k}{TB_{max}}$.

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.

3.3. 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:

$$B_k = \begin{cases} \frac{\mu_{C_k}}{T(1-S(\varepsilon_k))} & \text{if } \varepsilon_k < 1 - \frac{\mu_{C_k}}{B_{max}} \\ B_{max} & \text{if } \varepsilon_k \geq 1 - \frac{\mu_{C_k}}{B_{max}} \end{cases}. \tag{6}$$

If $\varepsilon_k > 1 - \frac{\mu_{C_k}}{TB_{max}}$, 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 5.

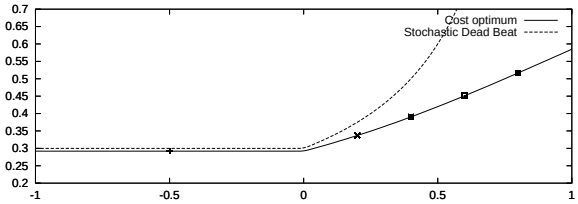


Figure 4. Optimal $B(\varepsilon)$ function for the optimal cost approach compared with SDB.

3.4. Optimal cost approaches

This technique is also based on the framework of stochastic control. In particular, taking inspiration from dynamic programming techniques [25], the controller chooses the value for the bandwidth B_k so as to optimise the expectation $\overline{w}(\varepsilon, b)$ (conditioned to the past evolution of the system) of a cost function $w(\varepsilon,\ b)$. 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 $\varepsilon_{k+1} = \varepsilon$.

In particular we chose a cost function accounting for the deviation of the next scheduling error from zero, and the bandwidth being used: $w(\varepsilon_{k+1}, b) = \gamma\varepsilon_{k+1}^2 + (1-\gamma)b$, where $\gamma \in]0, 1[$ allows us to decide different tradeoffs attaching more importance to the scheduling error and to the used bandwidth.

In case $\varepsilon_k = 1$, the minimum is immediately found as $B_k = \sqrt[3]{2\frac{\gamma}{1-\gamma}(\sigma^2+\mu^2)}$. In the other cases the following formula holds:

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

This formula can be directly used for all $\varepsilon_k \leq \varepsilon^* = 1 + \frac{3}{2\mu_C}\sqrt[3]{\frac{\gamma}{1-\gamma}(\sigma^2+\mu^2)}$, which is the range for which $\delta(\varepsilon_k)$ is real. As for the case of SDB, we used μ for the expectation of c_k, while σ denotes its standard deviation. For $\varepsilon_k > \varepsilon^*$, 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}. Both quantities are conditioned to the past evolution of the system and are the required output of the predictor for this control scheme.

Figure 4 reports the optimal $B(\varepsilon_k)$ function for a particular set of parameters. The same figure makes also a comparison with the bandwidth function in (6).

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 for the control

law:

$$B(\varepsilon_k) = \begin{cases} \frac{\sigma^2+\mu^2}{\mu[1-S(\varepsilon)]} & if\, \varepsilon < 1 - \frac{\sigma^2+\mu^2}{\mu B_{max}} \\ B_{max} & if\, \varepsilon_k \geq 1 - \frac{\sigma^2+\mu^2}{\mu B_{max}} \end{cases} . \quad (7)$$

It is easy to show that this solution is only valid if $B_{max} > \frac{\sigma^2+\mu^2}{\mu} = \mu + \frac{\sigma^2}{\mu}$. 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 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 deviation σ using larger bandwidth values to compensate for it.

4. Architecture of OCERA implementation

The applications to be scheduled with the techniques considered so far are soft real-time by nature. Thus it is of paramount importance to provide an implementation of such techniques in the context of a general purpose OS, where such applications are mostly used. We proved this to be feasible by providing a reference implementation in the Linux kernel, which is described in this section.

The implementation has been carried out in the context of the OCERA project, financially supported by the European Commission under the IST programme, fifth framework. The aim of the OCERA project is to enhance the real-time characteristics of Linux for both hard and soft real-time systems by providing a set of open software components. Although OCERA provides a high level of flexibility, allowing one to configure the system by selecting the appropriate components[2], in this paper we will focus on the soft real-time components, describing how adaptive reservations are implemented.

In OCERA, soft real-time tasks are implemented as regular Linux processes running in user space, and some scheduling modules are used to implement resource reservations as shown in Figure 5. The `QRES` module implements the resource reservation algorithm used for scheduling soft real-time tasks. The `QSPV` module provides is the *QoS Supervisor* that provides the admission control policy, whereas a set of QoS management modules implement the QoS control techniques described in Section (3). 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 RB scheduler) is separated from the *policy* implemented by the QoS modules. Moreover, both the scheduler and the controller are not *hardwired* 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 in the same system, each one serving a group of different tasks.

4.1. Generic Scheduler Patch

As previously said, the scheduling mechanism and bandwidth management policies are implemented by loadable modules. Such modules can customise the Linux scheduler's behaviour by using some symbols exported by the Generic Scheduler (gensched) patch. The gensched patch is a non-intrusive kernel patch that simply implements some hooks exporting them to loadable modules and it is a fundamental component of the OCERA architecture.

To better understand how our scheduler works, we will briefly recall here the structure of the Linux scheduler. The standard Linux kernel provides three scheduling policies: `SCHED_RR`, `SCHED_FIFO` and `SCHED_OTHER`. The first two policies are the "real-time scheduling policies", based on fixed priorities, whereas the third one is

[2]See http://www.ocera.org for a complete description

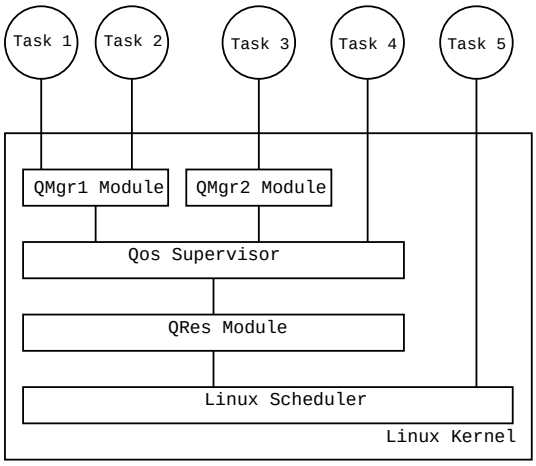


Figure 5. Structure of the soft real-time configuration of OCERA.

the default time-sharing policy. Linux processes are generally scheduled by the `SCHED_OTHER` policy, and can change it by using the `sched_setscheduler()` system call when they need real-time performance.

The Linux scheduler works as follows:

- The real-time task (`SCHED_RR` or `SCHED_FIFO`) having the highest priority is executed. If `SCHED_FIFO` is specified, then the task can only be preempted by higher priority tasks. If `SCHED_RR` is specified, after a time quantum (typically 10 milliseconds) the next task with the same priority (if any) is scheduled (with `SCHED_RR`, all tasks with the same priority are scheduled in round robin).
- If no real-time task is ready for execution, a `SCHED_OTHER` task can be executed.

Our scheduling module forces the Linux scheduling decisions by using the `SCHED_FIFO` policy and by setting the real-time priority of the selected task to the maximum possible value: in this way, no big modifications to the standard Linux scheduler are needed. To perform its scheduling decision, a scheduling module will need to intercept some relevant kernel events, select the task to be executed according to our customised policy, and raise its priority to the highest possible value. Then, the Linux scheduler will automatically be invoked when returning to user space and will dispatch the selected process.

The interesting events that we need to intercept are job arrivals, job finishings, process creations, and process terminations. In our framework, a job finishing corresponds to a task that blocks waiting for some event and a job arrival corresponds to a tasks that is unblocked. The gensched patch permits to easily intercept such events by exporting a set of function pointers, called *hooks*, that allow one to execute custom code segments when the events are triggered. For example, when a task blocks waiting for some event, the `block` hook is checked, and, if set, the corresponding function is invoked.

The most important hooks are: the `block` and `unblock` hooks, invoked when a task blocks and unblocks (job arrival and finishing), the `fork` and `cleanup` hooks, invoked when a task is created and terminated, and the `setsched` hook, invoked when the scheduling policy is changed. Initially, all hooks are unset in the kernel, and are not used.

Thus, modules that need to customise the scheduler behaviour, like our QRES module, may set the hooks to point to their appropriate handlers implementing the new behaviour. A more detailed description of the implementation of the generic scheduler patch and of the scheduling module can be found in [5].

4.2. Scheduling Module

The QRES scheduling module implements the CBS algorithm by Abeni and Buttazzo [3]: the CBS belongs to the class of the RB algorithms, described in Section 2.2, and is based on the earliest deadline first scheduler (EDF) [16]. In the original formulation, the CBS algorithm provides soft reservations; in our implementation we modified it to provide hard reservations (refer to [22] for a description of hard vs soft reservations). All the basic properties of the CBS scheduler still hold for our implementation, and the dynamical model described in Section 2.2 is still applicable. Since EDF is an optimal scheduling strategy on single processor systems, for the CBS algorithm theoretically U_{lub} is 1[3].

Once the QRES module has been loaded into the kernel, a task can require to be scheduled according to a RB policy by invoking the `sched_setscheduler()` system call with the policy `SCHED_CBS`. After such a call, the QRES hook handlers will intercept all scheduling events related to that task, implementing the desired scheduling policy. The `sched_setscheduler()` system call is also used by the task to specify scheduling parameters, through the use of an extended version of the structure `sched_param`. Specifically, the task is required to provide the desired budget and period into this structure.

The QRES implements the CBS algorithm by maintaining an internal queue of reservations ordered according to EDF. Whenever a `block` or `unblock` hook is invoked, the QRES module executes the proper rule of the CBS algorithm updating its internal variables and the EDF queue. Then, the first task in the queue is assigned the highest real-time priority, so that the Linux scheduler will dispatch it.

4.3. QoS Supervisor Module

Every time a certain bandwidth request is issued by a task, the system must check if there is enough free bandwidth to satisfy the request. This situation occurs when a new fixed reservation is created and every time a new value is required by the feedback mechanism of an adaptive reservation. This admission control is performed by the QoS supervisor module (denote with QSPV in Figure 5) upon every call of functions `request_qres_create`, `request_qres_change_budget`. Both functions are called by the handler of the `setsched` hook, which, in its turn, is called upon every invocation of the `scheduler_setsched()` system call.

Three different flavours 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 utilisations of the existing reservations, plus the utilisation of the new reservation, does not exceed 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 utilisation 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 [4, 2] 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.

[3] Actually, due to the overhead of the kernel, U_{lub} is slightly less than 1. In most practical experiments, U_{lub} was set equal to 0.98. However, in certain limit cases (for example when the period of the reservation is very short) the overhead is signifi cantly higher and U_{lub} can signifi cantly decrease.

```
int main() {
    // initialisation
    sched_param param;
    // init controller parameters
    qmgr_create(SCHED_QMGR1, &param);
    while (1) {
        //main loop code
        qmgr_end_cycle();
        wait_period();
    }
}
```

Figure 6. Typical structure of a cyclic task.

- 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.

4.4. The QoS Manager module

We provide different QoS management modules (denoted with `QMGR1` and `QMGR2` in Figure 5) that can coexist in the same system. Each module provides a different controller strategy and can serve more than one task. Indeed, different applications may need different controller strategies (in particular, the *Predictor* component in Figure 3 should be customised to the application needs). Each module will manage all tasks with the same characteristics, which presumably needs to be managed using 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.

A task using adaptive reservations must be linked with a `qoslib` library, which provides some commodity function implementing the user-level part of the feedback strategy. Recall that only periodic tasks are considered in this paper. As a result, a task attached to an adaptive reservation will have the structure shown in Figure 6.

The sequence of invocations is shown in the sequence diagram in Figure 7. In order to leave the kernel API unaffected by our changes, invocations to `qmgr_create` and to `qmgr_end_cycle()` use the `sched_setscheduler` system call to communicate with the kernel. At the beginning, the task must perform an initialisation 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 initialise the reservation budget and period. After initialisation, 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 the request with the new budget, the QoS supervisor can implement the three possible behaviours described above: *saturation*, *compression*, or *reject* (see [4, 2]). In any case, the `qspv_change_budget()` returns the actual value of the budget that has been set.

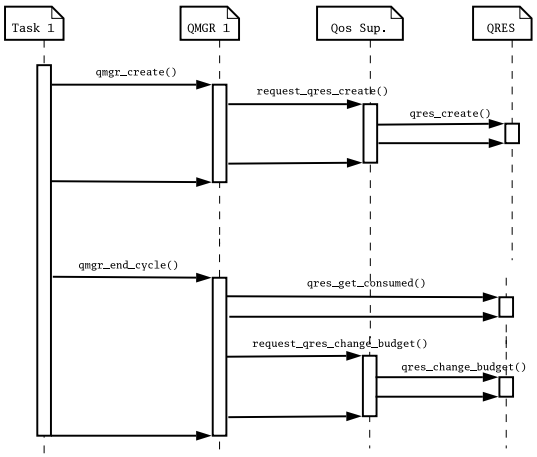


Figure 7. Sequence diagram that shows the interaction between the `QMGR`, the `QSPV` and the `QRES` modules.

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).

5. 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* [30] 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. The trace file, provided by Philips Research labs, refers to the same movie the segment in Figure 1 is taken from. We verified that the overhead introduced by the overall scheduling mechanism was sustainable to the point where we could perform the experiments on a slow machine (a first generation Pentium operated at 166MhZ).

In the first experiment, we show the benefit of adopting a feedback scheduling mechanism as opposed to a static allocation of the bandwidth. In the second set experiment we compared the performance of different controllers. Finally, in the third experiment, we evaluated the influence of the predictor component.

Benefits of feedback Consider again the MPEG decoding times shown in Figure 1. The scheduling error evolution achieved by a SDB controller is reported in Figure 8. The expected value μ_{C_k} is approximated by the predictor by performing a moving average of the last ten samples.

The only significant overshoot (around the 790th sample) is due to a swift scene change, which causes a temporary anomaly in predictor output. The problem is transient and it is soon recovered. A visual comparison between Figure 1 and Figure 8 is illustrative of the extent of the achieved improvement.

A more quantitative assessment can be made looking at experimental statistics reported in Table 1. The first three rows are referred to a statical allocation of bandwidth. For high values of the bandwidth, the mean scheduling error tends to negative values revealing a wasteful allocation. Notably the mean squared scheduling error - which is a metric encompassing both performance and efficiency - tends to increase with the bandwidth overallocation. On the contrary if the bandwidth is too small (as in the third row of the table) we may have a blow up of the

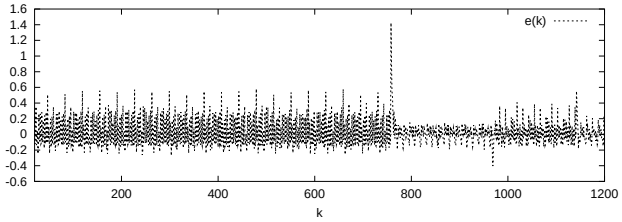


Figure 8. Scheduling error evolution resulting from the application of SDB controller to the input sequence of Figure 1(a).

Algorithm	Mean of ϵ_k	Std Dev of ϵ_k	Mean of ϵ_k^2	Mean of B_k
Static Alloc. (B=0.65)	1.845480	6.802540	49.680365	–
Static Alloc. (B=0.70)	-0.109443	0.196325	0.050522	–
Static Alloc. (B=0.75)	-0.189233	0.120037	0.050218	–
PI (poles in $0.9, 0.001$)	0.001980	0.138938	0.019308	0.643467
Optimal cost ($\gamma = 0.75$)	0.102481	0.070175	0.015427	0.607399
Invariant ($[-e, E] = [-0.16, 0.16]$)	0.014971	0.074434	0.005764	0.616828
SDB	0.004612	0.063865	0.004099	0.617666

Table 1. Quantitative performance evaluation based on experimental statistics.

mean squared scheduling error, meaning a remarkable degradation of the experienced Quality of Service (as it is pictorially shown in Figure 1.b). By varying the bandwidth between these extreme behaviours, it is possible to find the bandwidth value that minimises the mean squared error. The remaining rows of the table show the results that we obtained with feedback algorithms. In the first place, such algorithms have self-tuning abilities (we need not any prior knowledge of the application needs, as is the case for static allocation). The improvement in the mean squared scheduling error achieved by feedback scheduling algorithms with respect to the static allocation is absolutely evident. Moreover, it is remarkable that even in presence of a performance improvement of at least 50% all control algorithms use a mean bandwidth that is largely smaller than all static allocation reported in the table.

Comparing different controllers. In this section we compare the performance of four different controllers: 1) the switching PI controller [6, 17], 2) the invariant based controller [21], 3) the stochastic deadbeat controller, 4) the optimal cost controller. The experimental Probability Density Functions (PDF) resulting from the application of the four controllers are reported in Figure 9 and Figure 10. A more quantitative evaluation can be made looking at Table 1. A preliminary work was to *hand-tune* the parameters of the controllers so as to optimise their performance (in order to get a fair comparison).

As Figure 9 demonstrates, the switching PI controller is largely outperformed by both the invariant based and the stochastic dead beat schemes. The comparison between the SDB and the invariant based solution reveals very similar performance, and the impression is confirmed by the value of the mean squared error in Table 1. A strength of the SDB is its extreme simplicity, while the invariant based solution allows for more flexibility to the price of a greater design complexity.

In Figure 10 we compare the SDB and optimal cost approaches. Regarding the optimal cost controller, we report the results for two values of the γ parameter. In the first case we chose $\gamma = 0.90$, thus weighing very much the importance of the scheduling error. With respect to the SDB case, the PDF is shifted to the right; this is because

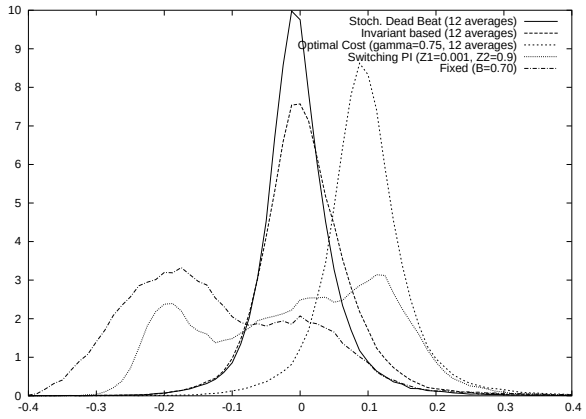


Figure 9. Experimental scheduling error PDF achieved by the stochastic based control schemes compared to the one achieved by the switching PI and a fixed bandwidth allocation.

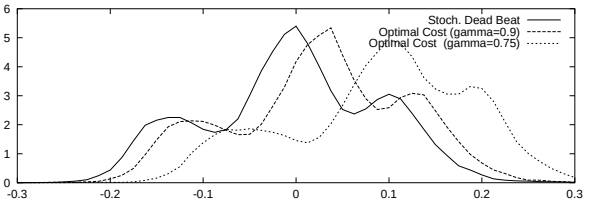


Figure 10. Experimental scheduling error PDF achieved by different stochastic based control schemes.

the optimal cost controller achieves a trade-off between bandwidth consumption and performance. Attaching even more importance to the bandwidth ($\gamma = 0.75$), the curve is further shifted to the right. Performance in terms of the scheduling error degrades, but the system tends to "save" bandwidth for other tasks (as it is possible to see in the last columnt of Table 1). As pointed out earlier, this flexibility is payed in terms of computational complexity. Finally, the performance of the algorithms along with the average used bandwidth is pictorially reported in Figure 11.

The importance of the predictor quality A first naive scheme for predicting computation times is based on a moving average of a certain number of samples for the computation times. This scheme does not take into account any peculiarity of the considered stochastic process. Actually, each application is characterised by peculiar properties on the input process that can be leveraged in order to improve the prediction, hence the closed-loop performance.

In our case we are dealing with an MPEG2 movie, consisting of a sequence of frames belonging to three classes I, P, B[28]. A common class of MPEG movies (typically used for DVD) has a periodic structure in the frame types, i.e. there exists a fixed sequence of frame types that repeats itself during the movie (a common example is $IBBPBBPBBPBBIBBP$...). The traces that we considered are derived from a DVD and have precisely such a structure (with periodicity 12). This is reflected in the autocorrelation function (see Figure 12), where peaks reveal that a I sample is highly correlated with the I sample of the next period, and so for the other frames in the period. Such a consideration suggests to use several moving averages in the predictor. A first possibility

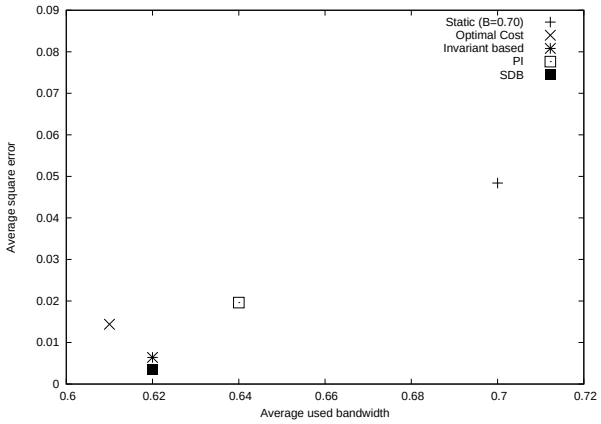


Figure 11. Graphical comparison of the performance of different feedback controllers.

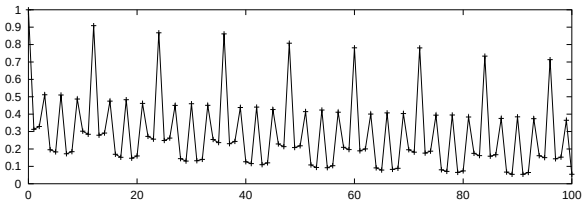


Figure 12. Autocorrelation function for the considered MPEG movie

is to use three moving averages: one for each class of frames. A second, more expensive, possibility is to use twelve moving averages (one for each position in the periodic sequences). Figure 13 compares the performance obtained by a SDB controller with the three types of predictors. As shown in the picture, the more sophisticated the predictor the narrower the resulting PDF, i.e. the more accurate the control results.

6. Conclusions and future work

In this paper, we addressed the problem of scheduling soft real-time systems by using reservation based scheduling techniques augmented by feedback control strategies.

We presented two important contributions to this problem. First, we proposed three new control strategies that are inspired to the ideas of stochastic control. We compared the performance of these new controllers with previous approaches.

Second, we described a software architecture for feedback control in the Linux operating system. The architecture has been implemented with a minimal set of modification to the Linux kernel. The software components are implemented as dynamically loadable kernel modules, and are organised according to a layered and modular structure. Thus the system is easy to modify and customise to different application needs.

In the next future, we will apply these techniques to different classes of applications, including an MPEG player, to demonstrate their effectiveness. Moreover, we want to consider alternative architectural solutions. An interesting possibility could be moving some of the QoS manager functionalities from kernel to user space. This would provide the QoS manager with additional flexibility than the one available in the kernel.

From a control theoretical point of view, we would like to attack the problem of scheduling tasks that use more resources (disk, network, etc.) in a coordinate way. In fact, a task may need different resources, other than the CPU, to complete its jobs. We could apply reservation based scheduling to other resources, but this allocation

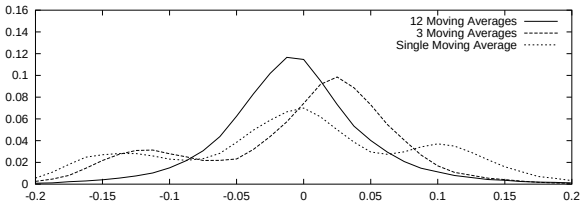


Figure 13. Scheduling error PDF obtained with a single moving average vs. 3 and 12 multiple moving averages.

must be coordinated in order to ensure a global QoS level.

References

[1] S. Abdelwahed, N. Kandasamy, and S. Neema. Online control for self-management in computing systems. In *Proc. of 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, Toronto, Canada, May 2004.

[2] L. Abeni. *Supporting time-sensitive Activities in a Desktop Environment*. PhD thesis, Scuola Superiore S. Anna, December 2002.

[3] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[4] L. Abeni and G. Buttazzo. Adaptive bandwidth reservation for multimedia computing. In *Proceedings of the IEEE Real Time Computing Systems and Applications*, Hong Kong, December 1999.

[5] L. Abeni and G. Lipari. Implementing resource reservations in linux. In *Proceedings of Fourth Real-Time Linux Workshop*, Boston, MA, December 2002.

[6] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole. Analysis of a reservation-based feedback scheduler. In *Proc. of the Real-Time Systems Symposium*, Austin, Texas, November 2002.

[7] G. T. C. Lu, J. Stankovic and S. Son. Feedback control real-time scheduling: Framework, modeling and algorithms. *Ppecial issue of RT Systems Journal on Control-Theoretic Approaches to Real-Time Computing*, 23(1/2), September 2002.

[8] A. Cervin, J. Eker, B. Bernhardsson, and K.-E. Årzén. Feedback-feedforward scheduling of control tasks. *Real-Time Systems*, 23(1), July 2002.

[9] F. J. Corbato, M. Merwin-Dagget, and R. C. Daley. An experimental time-sharing system. In *Proceedings of the AFIPS Joint Computer Conference*, May 1962.

[10] E. Eide, T. Stack, J. Regehr, and J. Lepreau. Dynamic cpu management for real-time, middleware-based systems. In *Proc. of 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, Toronto, Canada, May 2004.

[11] J. Eker. *Flexible Embedded Control Systems: Design and Implementation*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, 1999.

[12] G.Lipari and S. Baruah. Greedy reclaimation of unused bandwidth in constant bandwidth servers. In *IEEE Proceedings of the 12th Euromicro Conference on Real-Time Systems*, Stokholm, Sweden, June 2000.

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

[14] B. Li and K. Nahrstedt. A control theoretical model for quality of service adaptations. In *Proceedings of Sixth International Workshop on Quality of Service*, 1998.

[15] http://www.linux.org. Linux Offi cial Website.

[16] C. L. Liu and J. Layland. Scheduling alghorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1), 1973.

[17] C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. H. Son, and M. Marley. Performance specifi cations and metrics for adaptive real-time systems. In *Proceedings of the 21th IEEE Real-Time Systems Symposium*, Orlando, FL, December 2000.

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

[19] T. Nakajima. Resource reservation for adaptive qos mapping in real-time mach. In *Sixth International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, April 1998.

[20] L. Palopoli, L. Abeni, and G. Lipari. On the application of hybrid control to cpu reservations. In *Hybrid systems Computation and Control (HSCC03)*, Prague, april 2003.
[21] L. Palopoli, T. Cucinotta, and A. Bicchi. Quality of service control in soft real-time applications. In *Proc. of the IEEE 2003 conference on decision and control (CDC02)*, Maui, Hawai, USA, December 2003.
[22] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.
[23] D. Reed and R. F. (eds.). Nemesis, the kernel – overview, May 1997.
[24] J. Regehr and J. A. Stankovic. Augmented CPU Reservations: Towards predictable execution on general-purpose operating systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS 2001)*, Taipei, Taiwan, May 2001.
[25] S. Ross. *Introduction to stochastic dynamic programming*. Academic Press, 1983.
[26] http://www.aero.polimi.it/ rtai/. RTAI Offi cial Website.
[27] http://www.fsmlabs.com. FSMLabs - RTLinux Offi cial website.
[28] I. I. Standard. Iso/iec jtc1/sc29/wg11 mpeg-2: Generic coding of moving pictures and associated audio information, August 2000.
[29] D. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the Third usenix-osdi*. pub-usenix, feb 1999.
[30] http://xine.sourceforge.net. Xine Offi cial Website.
[31] R. Zhang, C. Lu, T. F. Abdelzaher, and J. A. Stankovic. Controlware: A middleware architecture for feedback control of software performance. In *Proc. of International Conference on Distributed Computing Systems*, Vienna, Austria, July 2002.