AQuoSA - Adaptive Quality of Service Architecture

Journal:	*Software: Practice and Experience*
Manuscript ID:	SPE-07-0099.R1
Wiley - Manuscript type:	Research Article
Date Submitted by the Author:	28-Jan-2008

Complete List of Authors:	Cucinotta, Tommaso; Scuola Superiore Sant'Anna, Real Time Systems Lab Palopoli, Luigi; University of Trento, DIT

SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **00**:1–7 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

AQuoSA – Adaptive Quality of Service Architecture

L. Palopoli*, T. Cucinotta*,†, L. Marzario†, G. Lipari†

ReTiS Laboratory, Scuola Superiore Sant'Anna, Via Moruzzi, 1, 56100 Pisa, Italy

SUMMARY

This paper presents an architecture for Quality of Service (QoS) control of time-sensitive applications in multi-programmed embedded systems. In such systems, tasks must receive appropriate timeliness guarantees from the Operating System independently from one another, otherwise the QoS experienced by users may decrease. Moreover, fluctuations in time of the workloads make a static partitioning of the CPU not appropriate nor convenient, whereas an adaptive allocation based on an on-line monitoring of the application behaviour leads to an optimum design.

By combining a resource reservation scheduler and a feedback based mechanism, we allow applications to meet their QoS requirements with the minimum possible impact on CPU occupation. We implemented the framework in AQuoSA [8], a software architecture that runs on top of the Linux kernel. We provide extensive experimental validation of our results and offer evaluation of the introduced overhead, which is perfectly sustainable in the class of addressed applications.

KEY WORDS: Resource Reservations; Adaptive QoS Control; Soft Real-Time; Embedded Systems

1. Introduction

Over the past years, real-time technologies, traditionally developed for safety-critical systems, have been applied within new application domains, including consumer electronics (e.g. cellular phones, PDAs), multimedia (e.g. video servers, VoIP gateways), telecommunication networks and others. Such applications are referred to as *soft real-time*, because, differently from traditional *hard real-time* ones, violation of their timing constraints does not cause a system

*Correspondence to: ReTiS Laboratory, Scuola Superiore Sant'Anna, Via Moruzzi, 1, 56100 Pisa, Italy
*University of Trento, Trento (Italy)
†Scuola Superiore Sant Anna, Pisa (Italy)
Contract/grant sponsor: FRESCOR European project; contract/grant number: FP6/2005/IST/5–034026

Received 13 April 1999
Revised 23 September 2002

failure, but rather a degradation in the provided Quality of Service (QoS). Such degradation often depends on the number and severity of constraints violations over a certain interval of time. Therefore, one goal in executing a soft real-time application is to keep under control timing constraint violations.

As an example, consider a video streaming application (e.g. DVD player or videoconferencing). One typical timing constraint is the necessity to process video frames at a regular frame rate. For example, for a rate of 25 frames per second (fps), every 40 msec the application is expected to load a frame from a source, decode it, apply some filtering, and show the frame on the screen. The application is usually designed to be robust to timing violations. For instance, the use of buffers allows to tolerate delays in decoding frames up to a maximum threshold, over which the users start perceiving a certain degradation of the provided quality of service (e.g. non equally spaced frames in time). Therefore, the user *satisfaction* is a decreasing function of the number and severity of timing violations.

The problem of providing QoS guarantees to time-sensitive applications has been traditionally faced with by using a dedicated device for each application. However, the flexibility of computer based devices can be exploited to full extent and with acceptable costs only if we allow for a concurrent utilisation of shared resources. For example, in a video-on-demand server, where many users can concurrently access the videos, it is important to provide a guaranteed level of QoS to each user. Even on a single-user terminal (e.g. desktop PC, PDA, mobile phone) it is important to provide stable and guaranteed QoS to different applications. Therefore, the problem arises of how to design a system that schedules accesses to the shared resources so that each application executes with a guaranteed QoS, and the resource is utilised to the maximum extent. Unfortunately, the scheduling solutions adopted in general purpose operating systems do not offer any kind of temporal guarantees.

A class of real-time scheduling algorithms, referred to as Resource Reservations (RR), has been proposed in the literature to provide the fundamental property of *temporal protection* (also called *temporal isolation*), in allocating a shared resource to a set of tasks that need to concurrently use it. Informally speaking, this means that each task is *reserved* a fraction of the resource utilisation, so that its ability to meet timing constraints is not influenced by the presence of other tasks in the system (a formal introduction to RR algorithms will be made in Section 2).

However, the availability of such a scheduling mechanism is not sufficient *per se* to ensure a correct temporal behaviour to the class of time-sensitive applications we are interested in. In fact, a fixed choice of the scheduling parameters for CPU allocation can be very difficult not only for the required knowledge of the computation time, but also due to its high fluctuations in time. A choice tuned on the expected average workload would lead to *temporary* degradations of the QoS, while a choice tuned on the worst-case workload would lead to a wasteful usage of resources. For this reason, the scheduling mechanism has to be complemented by a resource manager that can operate on the scheduling parameters[†].

[†] This paper does not deal with application-level adaptation, where applications may switch among various modes of operations.

1.1. State of the art

Different solutions have been proposed to schedule time-sensitive applications providing the temporal protection property, like the Proportional Share [40, 20] and Pfair [9] algorithms. They approximate the Generalised Processor Sharing (GPS) concept of a *fluid flow* allocation, in which each application using the resource marks a progress proportional to its weight. Similar are the underlying principles of a family of algorithms known as Resource Reservations schedulers [34, 4, 17, 35], that we will describe in Section 2.

As far as the problem of adaptive policies for QoS management is concerned, a remarkable work has been done in the direction of application-level adaptation [41, 42, 10]. A noteworthy solution is the one proposed in Q-RAM [33], that allows to associate resources to the different applications and generates a vector of recommended resource utilisations. This is, however, a static allocation approach due to the high computational requirements. However, it has been applied in the context of on-line schedulability tests on high-performance computers [15].

Concerning dynamic resource-level adaptation, a first proposal of this kind dates back to 1962 [12] and it applies to time-sharing schedulers. More recently, feedback control techniques have been applied to real-time scheduling [28, 36, 23, 11, 26] and multimedia systems [39]. All of these approaches suffer from a lack of mathematical analysis of the closed loop performance, due to the difficulties in building up a dynamic model for the scheduler. They typically apply classical control schemes (such as the Proportional Integral controller), with few theoretical arguments supporting the soundness of their design.

An approach similar in principles to Q-RAM is the one proposed in [1], where a polynomial (in the number of tasks multiplied by the number of resources) on-line algorithm is proposed that, based on a discrete integral controller running at periodic sampling instants, under the assumption of known lower and upper bounds to the derivative of the task consumption function (a QoS-level to resource requirement mapping) and its inverse, drives resources allocation towards achievement of fair QoS levels across all system tasks. Instead, we undertake the complementary approach to directly synthesise an ad-hoc non-linear controller based on a formal model for the system evolution and a formal statement of the control goals, providing conditions ensuring stability of the resulting closed-loop dynamics.

This is possible because, using Resource Reservations scheduling techniques, it is possible to derive a precise dynamic model of the system, whose closed-loop dynamics may be formally analysed. This idea, which we call *Adaptive Reservations*, was pioneered in [5]. In [18] a continuous state model of a Resource Reservations scheduler is used to design a feedback controller. In this case the objective of the controller is to regulate the progress rate of each task, which is defined associating a time stamp to an element of computation and comparing it with the actual time in which the computation is performed. A continuous state model for the evolution of a Resource Reservations scheduler is also shown in [7]. In this case, the authors propose the *virtual scheduling error* (i.e., the difference between the virtual finishing time and the deadline) as a metric for the QoS, which is adopted also in this paper. A control scheme based on a switching Proportional Integral controller is proposed in the paper and its performance is analysed in [29]. The problem is further investigated in [30] and in [6, 3], where deterministic and stochastic non-linear feedback control schemes taking advantage of the specific structure of the system model are shown. The idea for the controller is to use a

combination of a feedback scheme and of a predictor, and it is revisited in this paper (although in a different context). Approaches bearing a resemblance to the one just reported are shown in [2] and in [13], although in the latter paper most of the work is on the architectural side.

As far as architectural issues are concerned, some proposals performing the resource adaptation in the middleware are [43] and [13, 16]. The latter evolve around the QuO [21] middleware framework, which is particular noteworthy for it utilises the capabilities of CORBA to reduce the impact of QoS management on the application code. A CORBA-oriented approach may also be found in [38], where traditional control theory based on linearisations, proportional control and linear systems stability is applied to the context of controlling resources allocation for a real-time object tracking system.

Compared to previous work in the area (partly performed by some of the authors of this paper), we provide a set of innovative contributions. On the scheduling side, we introduce a tight yet simple mathematical model for the evolution of the QoS experienced by the tasks, that accounts for the allocation granularity imposed by the reservation scheduler (Section 3). This model allows to build a theoretically well-founded adaptive control law, split across a predictor and controller, acting locally on each task, where control goals are formally specified, and conditions for their achievement are formally identified (Section 4). We realised a modular architecture, where applications may choose the QoS controller that best suits their needs, or even provide their own controllers and/or predictors, if appropriate (Section 6). The algorithms may be compiled in user-space for maximum flexibility, portability and debugging capabilities, or in kernel-space for minimum overhead. A resource supervisor ensures global consistency of the system. Compared to other middleware layers [13, 16], our architecture presents a good degree of flexibility with a limited overhead (Section 6).

2. Resource Reservations

In this section we present the task model, and briefly describe the scheduler that will be used as a basis for the adaptive reservation framework.

The real-time task model A real-time task $\tau^{(i)}$ is a stream of jobs (or task instances). Each job $J_j^{(i)}$ is characterised by an arrival time $r_j^{(i)}$, a computation time $c_j^{(i)}$, and an absolute deadline $d_j^{(i)}$. When a job arrives at time $r_j^{(i)}$, the task is eligible for the allotment of temporal units of the CPU by the scheduler. The task executes for $c_j^{(i)}$ time units, then $J_j^{(i)}$ finishes at time $f_j^{(i)}$. We consider *preemptive* scheduling algorithms. In our model, job $J_j^{(i)}$ cannot receive execution units before $f_{j-\mathbf{1}}^{(i)}$, i.e. the activation of a job is deferred until the previous ones from the same task have been completed. Furthermore, we restrict to *periodic* tasks: $\tau^{(i)}$ generates a job at integer multiples of a fixed period $T^{(i)}$ and the deadline of a job is equal to the next periodic activation: $\forall j \geq 1$, $r_j^{(i)} = (j-1)T^{(i)}$ and $d_j^{(i)} = r_{j+\mathbf{1}}^{(i)} = jT^{(i)}$.

A real-time task $\tau^{(i)}$ is said to respect its deadlines if $\forall j, f_j^{(i)} \leq d_j^{(i)}$. We focus on *soft real-time tasks* that can tolerate deadline misses. Therefore, a job is allowed to complete after its

deadline, equal to the activation of the next job. In our model, when $J_j^{(i)}$ finishes, if $J_{j+1}^{(i)}$ has not yet arrived then $\tau^{(i)}$ blocks, otherwise $\tau^{(i)}$ is ready to execute.

The Constant Bandwidth Server Resource Reservations, originally proposed in [34], is a family of schedulers suitable for systems with hard, soft and non-real-time tasks. These algorithms provide the *temporal protection* property, thus it is possible to give *individual* guarantees to each task. Our framework is based on the Constant Bandwidth Server (CBS) [4]. While the applicability of the CBS is very general, we restrict for simplicity to the case of single threaded applications and periodic tasks.

In this case, each task $\tau^{(i)}$ is associated a *server* $S^{(i)}$ and a reservation $RSV^{(i)} = (Q^{(i)}, P^{(i)})$, with the meaning that, at time $kP^{(i)}$ from its start, the task has been scheduled for at least $kQ^{(i)}$ time units. $Q^{(i)}$ is the reservation *maximum budget* and $P^{(i)}$ is the reservation *period*. A server is a *schedulable entity* that maintains two internal variables, the current budget $q^{(i)}$ and the *scheduling deadline* $s^{(i)}$. A server is *active* at time t if its task has at least one active non-completed job at that time.

When a task $\tau^{(i)}$ becomes active due to the arrival of a new job at time t, the corresponding server deadline is set to $s^{(i)} \leftarrow t + P^{(i)}$, and the current budget is replenished to $Q^{(i)}$. The Earliest Deadline First (EDF) algorithm is used to schedule servers: among all active servers, the one with the earliest scheduling deadline is selected and the corresponding task is executed. Every time unit in which a task is executed corresponds to a decrement in the current budget of the associated server, until either the budget is exhausted or the task has completed all pending jobs, or it is preempted by another server with earlier deadline (for a complete description of the CBS algorithm, please refer to [4]). The CBS algorithm has been modified to implement both *hard* and *soft* reservations. In the former version, whenever a server exhausted the current budget, the associated task is suspended until the server deadline, when the current budget is recharged to $Q^{(i)}$. Therefore, task $\tau^{(i)}$ is guaranteed *exactly* $Q^{(i)}$ time units every $[kP^{(i)}, (k+1)P^{(i)}]$ time slot, and the processor may remain idle even in presence of pending jobs. In the latter version, instead, the current budget is immediately recharged and the server deadline postponed to $s^{(i)} \leftarrow s^{(i)} + P^{(i)}$, thus up to a time instant $kP^{(i)}$ from its start, a task is guaranteed *at least* $kQ^{(i)}$ time units.

By using EDF, the system can theoretically reach the maximum possible utilisation [24], under the global consistency constraint that the system must never violate:

$$\sum_i \frac{Q^{(i)}}{P^{(i)}} \leq U^{lub} = 1 \tag{1}$$

The ratio $B^{(i)} = \frac{Q^{(i)}}{P^{(i)}}$ is the *reserved bandwidth* and it can intuitively be thought of as the fraction of the CPU time reserved to the task.

The CBS algorithm satisfies the following properties, where $s^{(i)}(t)$ denotes the scheduling deadline of server $S^{(i)}$ at time t:

Theorem 1 (Temporal Protection [4]) *If Equation (1) holds, then at any time t, for each server $S^{(i)}$ active at time t, $t \leq s^{(i)}(t)$, i.e., each server always executes its assigned budget before its current scheduling deadline. Furthermore, if hard reservations are used, then it also holds: $t \geq s^{(i)}(t) - P^{(i)}$.*

Corollary 1. *Given a task $\tau^{(i)}$ handled by a server $S^{(i)}$, consider job $J_j^{(i)}$ with finishing time $f_j^{(i)}$ and let $s_j^{(i)}$ be equal to the latest server deadline during execution of $J_j^{(i)}$ (a.k.a. virtual finishing time): $s_j^{(i)} = s^{(i)}(f_j^{(i)})$. Then, in the soft reservation version, $f_j^{(i)} \leq s_j^{(i)}$, and in the hard reservation version, $s_j^{(i)} - P^{(i)} \leq f_j^{(i)} \leq s_j^{(i)}$.*

In the next section, we discuss how the above rules can be translated in a dynamic model for the execution of a task scheduled by the CBS.

3. Adaptive reservations

Adaptive Reservations are an extension of Resource Reservations addressing the problem of how to dimension the $(Q^{(i)}, P^{(i)})$ pair in presence of scarcely known and/or time-varying execution times. To this regard, a static choice for $(Q^{(i)}, P^{(i)})$ would lead to infeasible or inflexible solutions, while a feedback based mechanism can be used to self-tune the scheduling parameters and to dynamically reconfigure them in presence of changes in the workload. Such a mechanism may be defined in terms of: a *measured value*, used as input; an *actuator*, used to apply a corrective action whenever the measured value deviates from the desired value; a *dynamic model* of a reservation, useful for designing the controller. Hereinafter, we assume the use of hard reservations for the model and control design. Most results can be reformulated in a weaker form for soft reservations, but we omit them for the sake of brevity.

As adaptive reservations have the purpose of making the resource allocation continuously match the instantaneous workload, an ideal goal to pursue could be formalized as: schedule every task so that $\forall i, j, f_j^{(i)} = d_j^{(i)}$. This would guarantee not only respect of all deadlines, but also optimum occupation of the CPU by each task (i.e., minimum allocation sufficient to respect all deadlines). Therefore, the *scheduling error* $f_j^{(i)} - d_j^{(i)}$ is a reasonable choice as a measured value on which the feedback control scheme may work. When this quantity is positive (and this is allowed to occur in a soft real-time system), we need to increase the amount of CPU reserved to the task. When it is negative, then it means that the task received CPU time in excess and we may want to decrease it.

Unfortunately, using Resource Reservations, *it is impossible to control the exact time instant when a job finishes*, within the bounds identified by Corollary 1. Instead, we are able to control the *virtual finishing time*, therefore the *virtual scheduling error* defined as $\epsilon_j^{(i)} = s_{j-\mathbf{1}}^{(i)} - d_{j-\mathbf{1}}^{(i)}$, that may be regarded as a quantised measure of $f_{j-\mathbf{1}}^{(i)} - d_{j-\mathbf{1}}^{(i)}$. For the sake of brevity, from now on the term scheduling error will implicitly refer to the virtual scheduling error.

Concerning the choice of the actuator, we use the maximum budget $Q_j^{(i)}$ as an actuator variable (keeping constant the reservation period $P^{(i)}$), and we update it everytime we get a new measurement for the scheduling error, thus at each job finishing time $f_j^{(i)}$. In order to keep consistency of the CBS scheduler, the new maximum budget value can be applied only at the time of the next recharge for the server. Therefore, if the next job is already pending for execution at the time a job ends, the new job starts executing in the same server period as

the job that has just finished. The new job will use the residual budget in that server period first, then the new maximum budget computed by the controller for the subsequent periods.

3.1. Dynamic model of a reservation

As the discussion that follows is referred to a single task, for notational convenience we will drop the task index in this section and wherever convenient in the rest of the paper.

A considerable advantage of the choice of the measured value and actuator as proposed above is that it is possible to construct an accurate mathematical model of the system dynamic evolution. This model can be leveraged in the design of a feedback function. In order to construct such a model, we have to specify how the s_k variable evolves over time (consider Figure 1, showing a task τ with a period $T = 10$ that is assigned a reservation with period $P = T/2$ and an equal budget for the two jobs $Q_{\mathbf{1}} = Q_{\mathbf{2}} = 2$). We have to consider two cases: 1) $s_{k-\mathbf{1}} \leq d_{k-\mathbf{1}}$, 2) $s_{k-\mathbf{1}} > d_{k-\mathbf{1}}$. In the first case (Figure 1.a), job $J_{k-\mathbf{1}}$ finishes before the end of its period, so J_k becomes active right after its arrival and the number of server periods necessary to finish it is $\left\lceil \frac{c_k}{Q_k} \right\rceil$. Hence the last server deadline for J_k is given by: $s_k = r_k + \left\lceil \frac{c_k}{Q_k} \right\rceil P = d_{k-\mathbf{1}} + 2P$.

In the second case (Figure 1.b), jobs $J_{k-\mathbf{1}}$ and J_k share a reservation period, during which the system evolves with the previously assigned budget $Q_{k-\mathbf{1}}$. Let us introduce another state variable x_k, representing the residual budget of the shared server period usable by J_k when it is already pending at the time $J_{k-\mathbf{1}}$ finishes. If we also consider the possibility, for a new job starting within a shared reservation period with the previous one, to entirely fit within the residual available budget, then the following discrete event model is obtained:

$$
\begin{aligned}
x_{\mathbf{0}} &= 0 \\
x_{k+\mathbf{1}} &= \begin{cases} 0 & \text{if } s_k \leq d_k \\ x_k - c_k & \text{if } s_k > d_k \text{ and } c_k < x_k \\ Q_k \left\lceil \frac{c_k - x_k}{Q_k} \right\rceil - (c_k - x_k) & \text{if } s_k > d_k \text{ and } c_k \geq x_k. \end{cases} \qquad (2) \\
\epsilon_{k+\mathbf{1}} = s_k - d_k &= \begin{cases} \left\lceil \frac{c_k}{Q_k} \right\rceil P - (d_k - d_{k-\mathbf{1}}) = \left\lceil \frac{c_k}{Q_k} \right\rceil P - T & \text{if } \epsilon_k \leq 0 \\ s_{k-\mathbf{1}} - d_{k-\mathbf{1}} - (d_k & \end{cases}
\end{aligned}
$$

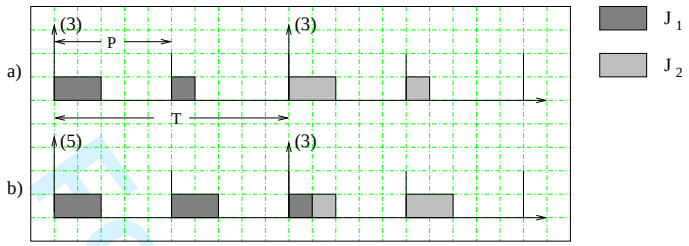


Figure 1. System evolution occurring when a job finishes within its deadline (a) and not (b).

where the computation time c'_k has been discounted of the disturbance x_k:

$$c'_k = \begin{cases} c_k & \text{if } \epsilon_k \leq 0 \\ 0 & \text{if } \epsilon_k > 0 \text{ and } c_k < x_k \\ c_k - x_k & \text{if } \epsilon_k > 0 \text{ and } c_k \geq x_k \end{cases}$$

Note that, replacing c'_k with c_k in Equation (4), the resulting model would represent the evolution of either an upper bound of the scheduling error, or the scheduling error itself obtained under the assumption that the residual computation time x_k in a reservation shared between two jobs is not utilised.

Hereinafter, we assume that $T^{(i)} = L^{(i)}P^{(i)}$ with $L^{(i)} \in \mathbb{N}^+$, i.e., the reservation period is an integer sub-multiple of the activation period. As a result, $\varepsilon_k^{(i)}$ takes values in the lattice set $\mathcal{E}^{(i)} = \{hP^{(i)},\ h \in \mathbb{N} \cap [-L+1, +\infty[\}$. We assume $Q_k^{(i)}$ to be a real number in the range $]0, P^{(i)}]$ and $c_k^{(i)}$ to be a positive real number, neglecting such issues as the quantisation of $c_k^{(i)}$ to the machine clock cycles.

3.2. Consistency of adaptive reservations

In this section we discuss when and how changes in Q_k do not compromise the consistency of the Resource Reservations scheduler. For the sake of simplicity, we assume that a change in the maximum budget Q_k is effective only from the very next recharge of the server budget[‡].

We define a global variable $B^{tot}(t)$ that keeps track of the total system bandwidth at all instants. Suppose that, at time t, a server $S^{(i)}$ needs to change its budget from $Q^{(i)}$ to $Q'^{(i)}$, and its bandwidth from $B^{(i)} = Q^{(i)}/P^{(i)}$ to $B'^{(i)} = Q'^{(i)}/P^{(i)}$. The following cases must be considered:

- $B'^{(i)} < B^{(i)}$: the new budget is applied from the next server instance. At the time the current server period ends $s^{(i)}(t)$, the total bandwidth can be decreased using the update

[‡] Actually it is possible to at least partially anticipate the change, but the resulting algorithm is more involved.

rule:

$$B^{tot}(s^{(i)}(t))_{new} = B^{tot}(s^{(i)}(t)) - B^{(i)} + B'^{(i)}. \quad (5)$$

- $B'^{(i)} > B^{(i)}$: we must distinguish two further cases:
  1. if $B^{tot}(t) - B^{(i)} + B'^{(i)} \leq U^{lub}$, then the total bandwidth is immediately increased by using the update rule in Equation (5), while the new budget is actually used starting from the next server instance;
  2. if $B^{tot}(t) - B^{(i)} + B'^{(i)} > U^{lub}$, then the system is experimenting an *overload condition* and the request cannot be directly accepted, because it would jeopardise the consistency condition of the CBS. It is then possible to operate with different policies, the simplest one consisting in *saturating* the request to $B^{(i)}(t) = U^{lub} - B^{tot}(t) + B^{(i)}$ (and we fall back in the previous case).

It is easy to show that, applying the rules described above, the properties of the CBS algorithm continue to hold. In particular, Theorem 1 and Corollary 1 are still valid.

Another possibility for managing overload conditions is to leave the request as pending, saturating it temporarily, then waiting for other servers to reduce their bandwidths. This way, whenever $B^{tot}(t)$ decreases as a result of another server decreasing its request, $S^{(i)}$ may increase its bandwidth to $\min\{B'^{(i)}, U^{lub} - B^{tot}(t) + B^{(i)}(t)\}$, thus increase the maximum budget accordingly from the subsequent recharge. This process of convergence of the current bandwidth $B^{(i)}(t)$ to the requested value $B'^{(i)}$ is greatly acceleretad if the other servers are forced to reduce their bandwidth occupation, for example due to a *fair rescaling* of the bandwidths among all the servers. In Section 4.4 we will describe the *supervisor* module, along with the exact policy it implements for managing this kind of situations.

As a final remark, Equation (4) describes accurately the evolution of the scheduling error assuming that the maximum budget is constant throughout a single job execution. However, as we have just seen, in case of overload the supervisor policy can change the bandwidth asynchronously with respect to job boundaries. In these conditions, the model in Equation (4) is still valid if the variable Q_k does not represent a *real* budget, but an *equivalent* one computed as an appropriate average on the sequence of budgets granted to a job (further details are omitted for the sake of brevity).

4. Feedback control mechanism

In this section we provide the control theoretical foundations of our approach. For the sake of brevity, we do not report the proofs of the theorems. The interested reader is referred to [31].

4.1. Design goals

The idea of a feedback controlled system typically subsumes two different facts: 1) the system exhibits a desired behaviour at specified equilibrium points, 2) the controller has the ability to drive the trajectories of the system state towards the desired equilibrium in a specified way (e.g., within a maximum time).

Due to the influence of the unpredictable c_k variable on the system evolution, our notion of equilibrium is forcibly restricted to the possibility for the controller to keep the system state within a prefixed range $\varepsilon_k \in \mathcal{E}$ around the origin, contrasting the action of the disturbance term. Also, for the kind of applications we are addressing, it is reasonable to rely on a last-minute estimate of the expected variability range for the unpredictable variable $c_k \in \mathcal{C}_k = [h_k, H_k]$ (we will clarify this concept later). A formal definition of this concept is offered next.

Definition 1. *Consider a system $\varepsilon_{k+1} = f(\varepsilon_k, c_k, Q_k)$ where $\varepsilon_k \in \mathcal{E}$ denotes the state variable, $Q_k \in \mathcal{Q}$ denotes the command variable, and $c_k \in \mathcal{C}$ denotes an exogenous disturbance term whose actual variability range $\mathcal{C}_k \subseteq \mathcal{C}$ is only known at time k and is constituted by a closed and bounded set $\mathcal{C}_k \in \mathcal{C}_C$. A set $\mathcal{I} \subseteq \mathcal{E}$ is said a* robustly controlled invariant set *(RCIS), if there exists a control law $g : \mathcal{E} \times \mathcal{C}_C \to \mathcal{Q}$, $q_k = g(\varepsilon_k, \mathcal{C}_k)$, such that, for all k_0, if $\varepsilon_{k_0} \in \mathcal{I}$, then $\forall k > k_0$ and $\forall c_k \in \mathcal{C}_k$, it holds that $\varepsilon_k \in \mathcal{I}$.*

In our case, the f function is defined by Equation (4) and the RCIS is an interval $[-e, E]$, with $E \geq 0$ and $(L-1)P \geq e \geq 0$, which we wish to be of minimum measure. Both extremal points of the interval, e and E, have a practical significance. By increasing E, we allow greater delays in the termination of each job, degrading the offered QoS level. On the other hand, increasing e we allow a task to terminate earlier, so it may receive more bandwidth than the minimum it strictly requires. As shown below, feasible choices for e and E result from the width of the uncertainty interval $\mathcal{C}_k$ and from the $\mathcal{Q}$ set.

The equilibrium condition can be sustained if: 1) the prediction of interval $\mathcal{C}_k$ is correct, 2) the value decided for Q_k is immediately and correctly applied. Occasional violations of these two conditions could drive the system state outside of the RCIS. In this case the desired behaviour for the controller is to restore the equilibrium situation. More formally:

Definition 2. *Consider a system as in definition 1 and two sets $\mathcal{I}$ and $\mathcal{J}$ with $\mathcal{I} \subseteq \mathcal{J} \subseteq \mathcal{E}$*

1. *$\mathcal{I}$ is* h-step-reachable *from $\mathcal{J}$, if there exists a control law $g : \mathcal{E} \times \mathcal{C}_C \to \mathcal{Q}$, $q_k = g(\varepsilon_k, \mathcal{C}_k)$, such that $\forall \varepsilon_k \in \mathcal{J} \backslash \mathcal{I} : c_j \in \mathcal{C}_j \forall j = k, \ldots, k+h \implies \varepsilon_{k+h} \in \mathcal{I}$*
2. *$\mathcal{I}$ is* reachable *from $\mathcal{J}$ if there exists h such that $\mathcal{I}$ is h-step-reachable from $\mathcal{J}$.*

We are interested in the case in which the target set $\mathcal{I}$ in the reachability definition is an RCIS for the system, whereas $\mathcal{J}$ defines the maximum deviation that a task is temporarily allowed to take from its desired condition (i.e. the maximum delay that it makes sense for a task to accumulate). If $\mathcal{J} = \mathcal{E}$, we speak of global h-step-reachability (or of global reachability).

We are now in condition to formally state design goals for the controller. Such goals are related to the QoS guarantees provided to each task $\tau^{(i)}$, for which we assume the definition of two intervals $\mathcal{I}^{(i)}$ and $\mathcal{J}^{(i)}$ with $\mathcal{I}^{(i)} \subseteq \mathcal{J}^{(i)}$ and of a number $h^{(i)}$. In particular, we partition the tasks in the system in the following three classes:

- **Class A**: 1) $\mathcal{I}^{(i)}$ is a globally reachable RCIS, 2) $\mathcal{I}^{(i)}$ is $h^{(i)}$-step-reachable from $\mathcal{J}^{(i)}$, 3) (ancillary) if the task features a piecewise constant computation time, then the $\epsilon^{(i)}$ is reduced to 0 in a finite number of steps.
- **Class B** (superclass of **class A**): $\mathcal{I}^{(i)}$ is globally reachable

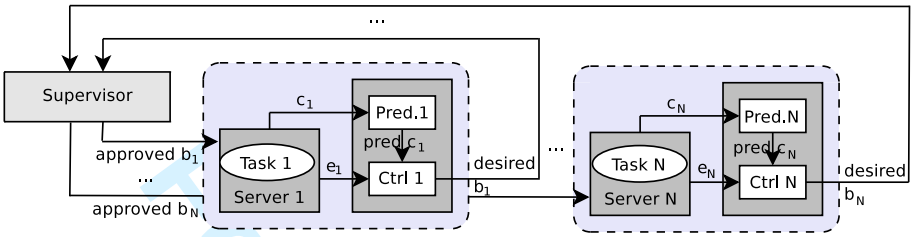


Figure 2. Control scheme: each task is attached a local feedback controller and the total bandwidth requests are mediated by a supervisor.

- **Class C**: no guarantee is offered but the controller tries to offer the same performance as for class A according to a best-effort policy (i.e., if there is availability of bandwidth).

The third goal for tasks of class A is an ancillary one: it refers to a limited class of applications, which have several operation modes and exhibit a repetitive behaviour in each mode, leading to a practically piecewise constant c_k.

4.2. Control architecture

Since we take measurements upon the termination of each job, we do not assume any fixed sampling period. In fact, our feedback scheme is based on a discrete event model. Moreover, a system-wide notion of "sampling" is entirely missing, since the sampling events are asynchronous for the different tasks. These considerations dictate a *decentralised* control scheme (Figure 2), where each task is attached a dedicated *task controller* that is responsible for maintaining the task QoS level within specified bounds with the minimum CPU utilisation. Still, the total bandwidth requests from the different task controllers is allowed to exceed the bound in Equation (1). To handle this situation (henceforth referred to as *overload*), a *supervisor* component is used to compute the actual bandwidth allocations that allow to respect the QoS requirements of the various classes of tasks.

One final requirement is that the workload due to the control components themselves needs to be very low. This rules out such design approaches as dynamic programming or model predictive control, which feature excellent performance at the price of unacceptable computation resources.

4.3. Task controller

A task controller is comprised of two components: a *feedback controller* and a *predictor*. At the termination of each job J_k, sensors located inside the Resource Reservations scheduler provide its computation time c_k and the experienced scheduling error ϵ_{k+1}. The former information may be used by the predictor in the estimation of a range $\mathcal{C}_{k+1}$ expected to contain the next

job computation time. This information, along with the measured ε_{k+1} value, is used by the feedback controller to fulfil the task design goals.

4.3.1. Feedback controller

In the following, we assume that the feedback controller operates with perfect predictions ($c_k \in \mathcal{C}_k$). It operates evaluating the worst case effects (with respect to the design goals) caused by the uncertainty of c_k. As discussed above, the reachability of the equilibrium makes the scheme resilient to occasional errors.

Robust controlled invariance Since ε_k evolves in the lattice $\mathcal{E}$ introduced in Section 4.1, the RCIS sets of interest are of the form $\mathcal{I} = [-e, E] = [-\hat{e}P, \hat{E}P]$, with $\hat{e} \in \mathbb{N} \cap [0, L]$, $\hat{E} \in \mathbb{N}$ (we chose $T = LP$). For the sake of brevity, we assume that $\hat{e} + \hat{E} < L - 1$. The following result provides what choices of $\hat{e}$ and $\hat{E}$ yield an attainable RCIS, for a given prediction $\mathcal{C}_k = [h_k, H_k]$ and saturation value for the command variable $\overline{Q}$, and what feedback control laws enact robust controlled invariance of $\mathcal{I}$.

Theorem 2. *Consider system in Equation (4) and assume that $c_k \in \mathcal{C}_k = [h_k, H_k]$, and $Q_k \in \mathcal{Q} = [0, \overline{Q}]$. Consider the interval $\mathcal{I}$ as defined above and let $\alpha_k \triangleq \frac{h_k}{H_k}$, $\alpha \triangleq \inf_k\{\alpha_k\}$, $H \triangleq \sup_k\{H_k\}$. Then, $\mathcal{I}$ is a RCIS if and only if*

$$\hat{e} + \alpha\hat{E} > L(1-\alpha) - 1 \wedge \overline{Q} > \frac{H}{L}, \tag{6}$$

furthermore, the family of controllers ensuring robust controlled invariance of $\mathcal{I}$ is described as follows:

$$Q_k \in \left[\frac{H_k}{L + \hat{E} - \frac{S(\varepsilon_k)}{P}}, \min\left\{ \frac{h_k}{L - 1 - \hat{e} - \frac{S(\varepsilon_k)}{P}}, \overline{Q} \right\} \right[\tag{7}$$

where $S : \mathbb{R} \to [0, +\infty[$ is defined as $S(x) \triangleq \max\{0, x\}$.

The result above does not simply lead to a feedback controller but to a family of feedback controllers that can keep the system state in the desired equilibrium. Possible guidelines for feedback design can be:

- choose the leftmost extremal point in Equation (7) to either save bandwidth or have the maximum possible tolerance to small violations of the lower prediction bound of the prediction ($c_k < h_k$);
- choose the rightmost extremal point to have the maximum possible tolerance to small violations of the upper prediction bound $c_k > H_k$ for the next sample;
- choose the middle point in order to gain maximum robustness with respect to violations of both bounds;
- choose a point which, for piecewise constant inputs, realises a perfectly null scheduling error as soon as possible;
- choose a point which optimises some other property of the system, such as a cost function defined by weighting the resulting uncertainty on the next scheduling error and the used bandwidth.

Remark 1. *It is straightforward to show that if the predictor succeeds with probability at least p ($\Pr\{c_k \in \mathcal{C}_k\} \geq p$), then the control approach proposed above ensures that $\Pr\{\varepsilon_{k+1} \in \mathcal{I} \mid \varepsilon_k \in \mathcal{I}\} \geq p$. In simpler words, controlled invariance of $\mathcal{I}$ is preserved with a probability at least equal to the one of producing a correct prediction.*

4.3.1.1. Reachability The following result shows how to steer the system from a state belonging a set of the form $\mathcal{J} = [-(L-1)P, \Gamma P]$, with $\Gamma > \hat{E}$, back into an RCIS interval of the form $\mathcal{I} = [-(L-1)P, \hat{E}P]$, with $\hat{E} \in \mathbb{N}$. In view of Equation (4), if the system is able to preserve invariance of $\mathcal{I}$, then it is trivial to steer the system from a negative scheduling error to $\mathcal{I}$ in one step. Therefore, bounding e to the maximum possible value of $(L-1)P$ is not a loss of generality.

Theorem 3. *Consider the system defined by Equation (4). Consider the intervals $\mathcal{I}$ and $\mathcal{J}$ as defined above. Let $\tilde{H}_h = \sup_k \frac{1}{h} \sum_{j=k}^{k+h-1} \left\lceil \frac{H_j}{Q} \right\rceil \overline{Q}$ and assume that limit $\tilde{H} \triangleq lim_{k \to +\infty} \frac{1}{k} \sum_{j=0}^{k-1} \left\lceil \frac{H_j}{Q} \right\rceil \overline{Q}$ exist and be finite. Then:*

1. *a sufficient condition for the global reachability of $\mathcal{I}$ is $\overline{Q} > \frac{\tilde{H}}{L}$;*
2. *a necessary condition for the global reachability of $\mathcal{I}$ is $\overline{Q} \geq \frac{\tilde{H}}{L}$;*
3. *if $\mathcal{J}$ is globally reachable, then $\mathcal{I}$ is h-step-reachable from $\mathcal{J}$ if and only if $h \geq \left\lceil \frac{\Gamma - \hat{E}}{L-1} \right\rceil$ and $\overline{Q} \geq \frac{h\tilde{H}_h}{hL - \Gamma + \hat{E}}$;*

4.3.1.2. Control to zero Assume that the system evolves in a RCIS using a control law compliant with Equation (7). Concerning the third requirement for class A tasks, the following result shows under what conditions the error may be reduced to 0 in one step, whenever the controller has a good estimate of the next computation time (as it might happen if the input features a piecewise constant behaviour).

Fact 1. *Assume that the system evolves in a RCIS $\mathcal{I} = [-e, E]$ using a control law compliant with Equation (7). Let $H = \sup_k H_k$ and $h = \inf_k h_k$. Then, the set $\mathcal{Z}$ of c_k values which, if known in advance by the controller, allow a choice of Q_k among the ones dictated by Equation (7) which controls the scheduling error to zero in one step $\forall \varepsilon_k \in \mathcal{I}$, is given by:*

$$\mathcal{Z} = \left\{ \tilde{c} \mid H\frac{T-E-P}{T} < \tilde{c} \leq h\frac{T}{T-P-e} \wedge \tilde{c} \leq \frac{\overline{Q}}{P}(T-E) \right\}, \tag{8}$$

where the 1-step zero controller may choose Q_k in the following range $q_k \in \left[\frac{\tilde{c}}{T - S(\varepsilon_k)}, \frac{\tilde{c}}{T - S(\varepsilon_k) - P} \right[$ intersected with Equation (7).

4.3.2. Predictor

Due to Theorem 2 and Remark 1, the prediction interval should be as small as possible, so to have an RCIS as small as possible. On the other hand, it is important that it be correct with a good probability, so to have a good probability of keeping the scheduling error within the

RCIS. Therefore, it is necessary to find a good trade-off between a tight interval and a good prediction probability. This design activity is influenced by the knowledge on the stochastic properties of the process $\{c_k\}$, which are largely application dependent. For this reason our architecture (see Section 5) allows the application to use a custom predictor together with the task controller. This is extremely useful for exploiting results like found in [37], where it is shown that, while parsing an MPEG2 or MPEG4 video frame, from such information as the pixel count, byte count, macroblock counts and a few others, it is possible to build an accurate estimate of the frame decoding time with roughly a 5% overhead, a great part of which due to the frame parsing operation that extracts the cited parameters, and that is anyway needed by the decoder for its operation. Nonetheless, the architecture provides a set of simple "library" predictors of general usefulness, displaying a fair level of performance for a wide set of applications, at a very limited computational cost.

Considering the set of the latest n observed computation times, the prediction interval $\mathcal{C}_k$ is obtained as a weighted average of these samples $\mu_k = \sum_{j=1}^{n} w_j c_{k-j}$ plus/minus their standard deviation σ_k multiplied by a constant γ, that permits to trade prediction accuracy against probability of wrong estimation. If the weights are all equal to $1/n$, we get a standard Moving Average of length n, henceforth denoted as MA[n]. In this case, μ_k and σ_k may be computed with a $O(1)$ computation time and a $O(n)$ memory occupation.

In some cases (e.g. MPEG decoding), the application has a periodic pattern (e.g. due to the Groups of Pictures), which is reflected in the autocorrelation structure of the c_k process. In this case, we experimented using more than one moving average, e.g. as many moving averages as the periodicity of the process. If S is the period of the sequence, this algorithm features a $O(1)$ computation time and a memory occupation of nS. We denote this type of predictor as MMA[n,S], meaning that it runs S moving averages of length n each.

The algorithms proposed above are very efficient and offer an acceptable accuracy (see Section 6). Also, we evaluated the possibility to compute the taps w_j as a result of a least square optimisation program. If the process stochastic properties do not change too much in time, it is possible to have a first "training" phase, during which the application executes with a fixed bandwidth and a certain number of samples are collected. When the number of samples is sufficient, it is possible to optimise the prediction filter and - henceforth - proceed with the computed taps. We will denote this solution as OL[n,N], where n denotes the number of taps and N the length of the training set. More sophisticated prediction schemes (e.g. adaptive ones like recursive least squares) lead to an unsustainable overhead.

4.4. Supervisor

The main role of the supervisor is to ensure that each task receives its guaranteed amount of bandwidth, whenever its dedicated task controller requires it, even when the system is overloaded and the cumulative requests of bandwidth from the different controllers exceed U^{lub}. To this regard, Theorem 3 provides an estimation of the minimum bandwidth $\overline{B}_B^{(i)} = \overline{Q}_B^{(i)}/P^{(i)}$ required for tasks of class B. Likewise, the minimum bandwidth $\overline{B}_A^{(i)} = \overline{Q}_A^{(i)}/P^{(i)}$ required for tasks of class A is the maximum between the one indicated in Theorem 3 and the one indicated in Theorem 2. Note that for the same task $\tau^{(i)}$, $\overline{B}_A^{(i)} \geq \overline{B}_B^{(i)}$.

In order for the task controllers to attain their design goals, the supervisor has to behave as follows:

- if task $\tau^{(i)}$ is of class A, then if $B^{(i)} \leq \overline{B}_A^{(i)}$ then $B^{(i)}$ is granted; otherwise the supervisor has to grant at least $\overline{B}_A^{(i)}$;
- if task $\tau^{(i)}$ is of class B, then if $B^{(i)} \leq \overline{B}_B^{(i)}$ then $B^{(i)}$ is granted; otherwise the supervisor has to grant at least $\overline{B}_B^{(i)}$;
- the difference between U^{lub} and the total bandwidth that has to be granted according to the first two points is allocated to the tasks using some heuristics (see Section 5.3).

Clearly, a preliminary condition is that the relation between the running tasks holds:

$$\sum_{j \in \text{class A}} \overline{B}_A^{(j)} + \sum_{j \in \text{class B}} \overline{B}_B^{(j)} \leq U^{lub} \tag{9}$$

As a final remark, the changes in the bandwidth of the tasks subsequent to the occurrence of an overload condition have to be performed according to the rules stated in Section 3.2 to maintain consistency of the adaptive reservation mechanism. Therefore the configuration of the scheduler eventually decided by the supervisor is not instantly applied. However, in our practical experience, the feedback scheme is resilient to the small delays thus introduced.

5. Software architecture

In this section we describe the software architecture we developed for providing a concrete implementation of the techniques proposed in the previous sections. As a reference system, we chose the GNU/Linux OS, with kernel versions up to 2.4.32 and 2.6.21. A considerable advantage of Linux, alongside of the free availability of source code and documentation, is its modular structure that allows to extend the kernel functionality by simply inserting a module. Our software is comprised of a set of application libraries and of dynamically loadable kernel modules.

5.1. Design goals and architecture overview

The design of the system was carried out pursuing the following goals:

- Portability: the link between the proposed architecture and the adoption of a specific kernel platform is shallow. To achieve this goal, we designed a layered structure where kernel-dependent code is confined inside the lowermost level. Moreover, the changes made on the kernel are minimal and the communication between the different components of the architecture (which runs partly at user and partly at kernel level) uses virtual devices, which are commonplace in POSIX Operating Systems.
- Backward compatibility: existing non real-time applications run without modifications and are scheduled by the standard Linux scheduler in the background w.r.t. soft RT applications.

Figure 3. System Architecture.

- Flexibility: our architecture allows to easily introduce new algorithms, different from those shown in this paper. These algorithms can be run either in user or in kernel space.
- Efficiency: the overhead introduced by our framework, tion to the one due to context switches (that are a direct consequence of the new functionality) is negligible;
- Security: in order to tacks based on the new mechanism, it must be possible to define maximum CPU bandwidths on a per-user or per-group basis (in the same way as disk quotas).

The proposed architecture is depicted in Figure 3, and it is composed of the following main components:

- the Generic Scheduler Patch (GSP), a small patch to the kernel (less than 200 lines in 7 modified files) which allows to extend the Linux scheduler by intercepting scheduling events and executing external code in a kernel module;
- the Kernel Abstraction Layer (KAL), a set of C macros that abstract the additional functionality asure time and set timers, ability to associate data to the tasks, etc...;
- the QoS Reservation component, composed of a kernel nd of an application library communicating through a Linux virtual device:
  - the Resource Reservation module implements an EDF scheduler, the resource reservation mechanism (based on EDF scheduler) and the RR supervisor; a set of compile-time configuration options allows one to ifferent Resource Reservation (RR) primitives, and to customise their exact semantics (e.g. soft or hard reservations);

  - the Resource Reservation library provides an API allowing an application to use resource reservation functions;

- the QoS Manager component, composed of a kernel module, an application library, and a set of predictor and feedback sub-components which may be configured to be compiled either within the library or within the kernel module. It uses the RR module to allocate the CPU:
  - the QoS Manager module offers kernel-space implementations of the feedback control and prediction algorithms shown in this paper;
  - the QoS Manager library provides an API allowing an application to use QoS management functionality; as far as the control computation is concerned, the library either implements the control loop (if the controller and predictor algorithms are in user-space) or redirects all requests to the QoS Manager kernel module (in case a kernel-space implementation is required). The latter communicates with the Resource Reservation module to take measurements of the scheduling error or to require bandwidth changes. Consistently with the feedback scheme presented in the previous section, such requests are "filtered" by the QoS supervisor.

A detailed description of the different components follows.

5.2. Generic Scheduler Patch

A preliminary description of this patch can be found in [22]. The idea is not to implement the Resource Reservations by a direct modification of the Linux scheduler. Instead, the Generic Scheduler Patch (GSP) intercepts scheduling related events invoking appropriate functions inside the Resource Reservation module. This way, it is possible to force Linux scheduling decisions without replacing its scheduler (which can be called, for instance, to schedule the non real-time activities in background).

The patched kernel exports a set of function pointers (called `hooks`). The code excerpt in Figure 4 clarifies this concept: whenever a scheduling-related event occurs, the appropriate function is invoked through a function pointer (the hook), if set. The table of hooks may be appropriately set by a dynamically loadable module[§]. The relevant events for which hooks have been introduced are: task creation and termination, block and unblock when accessing a shared resource, stop and continue due to receive of the SIGSTOP and SIGCONT signals. For each of these events, there is a corresponding pointer in the hook table: `fork_hook`, `cleanup_hook`, `block_hook`, `unblock_hook`, `stop_hook` and `continue_hook`.

The event handlers and the necessary data structures (e.g., scheduling queues) are contained in a kernel module. They receive the pointer(s) to the `task_struct` of the interested task(s)

[§]Due to the full preemptiveness of the latest kernel releases, use of hooks in the shown code fragment and changes to the hook table need to follow an appropriate synchronisation protocol, whose details are omitted for the sake of brevity.

```
/* Code in the patched kernel */

/* Modification of the Linux task struct */
struct task_struct {
  /* Linux standard fields */
  ...
  void *scheduling_data;              /* GSP additional field     */
};

/* Definition of the hook */
void (*fork_hook)(void *) = NULL;

/* Standard Linux function executed upon the fork */
... do_fork (...) {
  if ((fork_hook!=NULL) && is_realtime(task))
    fork_hook(task);
  /* Original code of do_fork() */
    ...
}
```

```
/* Code in the scheduling module */

extern void (*fork_hook)(void *);

/* Definition of the handler */
void fork_hook_handler(void *) {...};

/* Standard Linux module initialisation function */
... init_module(..) {
  ...
  fork_hook = fork_hook_handler;
  ...
}
```

Figure 4. Example utilisation of the hook mechanism.

as parameter(s). The GSP patch also allows to link external data to each task, through a new `scheduling_data` field that extends the `task_struct` structure. As an example, the Resource Reservation module implementing the Resource Reservations uses this pointer to quickly associate each task to the server it is currently running into.

5.3. QoS Reservation component

To better understand how our scheduler works, we briefly recall here the structure of the Linux scheduler. The standard 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 the default time-sharing policy. Linux processes are generally scheduled by the `SCHED_OTHER` policy, but privileged tasks can change it by using the `sched_setscheduler()` system call when they need real-time performance.

The Linux scheduler works as follows:

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

Furthermore, Linux implements POSIX threads as tasks that share a set of resources (e.g. the memory tables), but the scheduler is only concerned with scheduling of single threads of execution, referred to as *tasks* for historical reasons. Therefore, from now on we will use the term task to denote either a single-threaded process, or a single thread within a multi-threaded process.

The KAL layer affects the Linux scheduler decisions by simply manipulating the queue(s) of ready tasks used by the standard kernel scheduler, with the help of the GSP patch functionality. Tasks are forced to run at a higher priority than any other Linux task by assigning them a `SCHED_RR` policy and a statically configurable real-time priority. On the contrary, tasks are forbidden to run (for example for implementing hard reservations) by temporarily removing them from the Linux ready queue (for kernel 2.4 we used to set their real-time priority to a value below the minimum possible value).

The described mechanism works as expected as long as there are no Linux tasks running at a real-time priority higher than the one used by the KAL, and the soft real-time tasks themselves do not try to use Linux real-time scheduling facilities, in addition to the functionality provided by our infrastructure. In the former case, the reserved tasks would undergo the interferences of the higher priority tasks. In the latter case, their requests to the Linux scheduler would be silently ignored by the kernel.

This approach minimises the changes required to the kernel and it permits coexistence and compatibility with other patches that modify the scheduler behaviour, e.g. the preemptability patch [32].

We are now ready to present our QoS Reservation component. The core mechanism is implemented in a Linux kernel module: the Resource Reservation module. The scheduling service is offered to applications through a library (Resource Reservation library). The communication between the Resource Reservation library and the Resource Reservation module is established through a Linux virtual device.

Resource Reservation module This module implements various resource reservation algorithms: CBS [4], IRIS [27] and GRUB [17]. When the module is inserted, the appropriate hooks are set and the data structures are initialised (see Figure 4). Linux tasks that do not need resource reservations are still managed by the Linux scheduler. Tasks that use the Resource Reservation mechanism are scheduled by our module. The module internally implements an EDF queue with the purpose of implementing the selected Resource Reservations algorithm (CBS or one of its variants).

The module is configurable to allow a server to serve only one task or, alternatively, a group of tasks. The second choice is particularly useful when allocating bandwidth to a multi-threaded application since developers/designers do not need to allocate bandwidth to each thread but to the entire application. This option, however, is not illustrated in depth in this paper, where we assume that a server is only used for one task.

If the module is configured for allowing multiple tasks per server, one default server is dedicated to Linux tasks that are not associated to any server. This is to avoid that reserved tasks starve standard Linux tasks and system services.

QoS supervisor component The bandwidth requests coming from the task controllers can be accepted, delayed, reshaped or rejected by the QoS Supervisor, in order to enforce the global consistency relation in Equation (1). In addition, for security reasons, we want to avoid that a single user, either maliciously or due to a software bug, causes a system overload by requesting too much bandwidth and forcing tasks of other users to get a reduced bandwidth assignments. For this reason, a privileged user (e.g. the system administrator) can define maximum bandwidths for each user and users group. The supervisor operates both at the creation of a server and during the execution of a task. In the first case, the application provides the required minimum guaranteed bandwidth, that undergoes the admission control process, based on the test in Equation (9). When the system is not overloaded, the QoS Supervisor propagates (see also Section 4.2) to the scheduler the bandwidth requests from the task controllers. In overload conditions, tasks of class A and B receive at least their minimum guaranteed bandwidth if they required it $\overline{B}_A^{(i)}$ and $\overline{B}_B^{(i)}$. The bandwidth left from this initial assignment $U^{lub} - \sum_{i \in classA} \overline{B}_A^{(i)} - \sum_{i \in classB} \overline{B}_B^{(i)}$ is assigned using a heuristic based on *priority levels* and *weights*. First, requests from tasks with higher priority levels are satisfied. Among tasks in the same level, the bandwidth is shared in proportion to their weights. Moreover, maximum per-level bandwidth limits may be specified so as to avoid complete starvation of lower levels.

The supervisor policy, based on the different configurations options described earlier, can be configured into the system by a privileged user by means of the Resource Reservation library. The administrator can configure the supervisor through a configuration file.

Resource Reservation library The Resource Reservation library exports the functionality of the Resource Reservation module and of the QoS Supervisor module to the user applications through an appropriate API. All functions can operate on any Linux process, provided that the application has enough privileges.

The library provides an API to:

- create a new server with either statically or dynamically assigned bandwidth;
- change and retrieve server parameters after creation;
- move a task from a server to another;
- configure the supervisor.

The communication with the Resource Reservation module and the QoS Supervisor module is performed through a Linux virtual device and the `ioctl()` primitive.

5.4. QoS manager component

This component is responsible for providing a task with a set of standard control and prediction techniques that may be used by the task in order to dynamically adjust its bandwidth requirements.

The QoS Manager is split into two distinct parts: a user-space library and a loadable kernel module. The control and prediction algorithms can be located in both spaces. The application is always linked to the QoS Manager library and interacts with it. Depending on the configured location of the required sub-components, the library redirects all library calls to either user-space code or kernel-space management code (through the proper virtual device).

In Figure 5 we report a sequence diagram that shows the sequence of functions called when a job ends if the controller is implemented as a library function in user-space (the case of kernel-space implementation is very similar).

When a job of a periodic task is about to suspend, it calls `qmgr_end_cycle()` which is redirected to the appropriate function (for kernel-space implementation, the primitive is translated into a `ioctl()` invocation to the virtual device).

The main difference between user space and kernel-space implementation is in the number of times the application switches between user and kernel-space. For kernel-space implementation

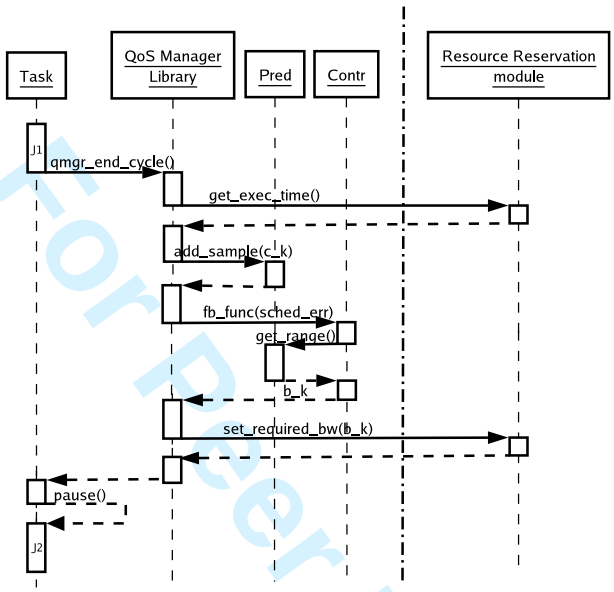


Figure 5. Interaction among various parts of the architecture when controller and predictor are in user space. The dash and dot line separates between user-space and kernel-space components.

program within a server, so that any spawn thread or task is automatically attached to the same server.

6. Experimental Results

In order to validate the presented approach, we used our middleware in an example application consisting of a simple MPEG-2 decoder, which complies with the periodic task model shown earlier. The implementation was based on the FFMPEG library [14] and it consists of a task that periodically reads a frame from a RAM virtual disk partition, and decodes it writing the result into the frame-buffer. The decoding task has a period of 40 ms (corresponding to a standard 25f/s).

We performed two classes of experiments. The first one is aimed at showing the effectiveness of the feedback controller, by comparing the evolution of the scheduling error attained for different configurations. The second one is aimed at assessing the overhead introduced by our mechanisms. The description of the hardware/software platform we used is in Table I.

Experiments have been done on a system running the Linux 2.4.27 kernel release. Together with the GSP patch, we applied two additional patches: the High Resolution Timer (hrtimers-2.4.20-3.0; [19]) and Linux Trace Toolkit (TraceToolkit-0.9.5; [25]).

```
/* Task structure */

void my_task() {
  /* Initialise QoS Manager and check if modules loaded */
  if (qmgr_init() != QOS_OK) { /* Handle error */ }

  /* Set controller parameters */
  qmgr_set_task_period(TASK_PERIOD);
  qmgr_set_server_period(SERV_PERIOD);
  qmgr_set_max_bandwidth(SERV_MAX_BW);

  /* Set predictor and controller parameters */
  qmgr_set_predictor(QMGR_PRED_MOVAVG);
  movavg_set_sample_size(DEFAULT_SAMPLE_SZ);
  qmgr_set_controller(QMGR_CTRL_INVARIANT);

  /* Performs the admission control */
  if (qmgr_start() == QOS_OK) {
    /* job executions (main loop) */
    while (condition) {
      do_job();

      /* takes measurements and performs feedback control */
      qmgr_end_cycle():

      /* go to sleep waiting for next activation */
      wait_for_next_job();
    }
  } else {
    /* Notify to user overload condition */
  }
  /* Release data structures and stop feedback loop */
  qmgr_end();
}
```

Figure 6. Sample task code

The use of the High Resolution Timers patch is *de facto* necessary for our applications since the default Linux timers resolution is too low for our purposes (10 ms for kernel 2.4, may be configured up to 1 ms for kernel 2.6). Using the High Resolution Timers patch, it is possible to set timers and take time measurements with a granularity in the order of ten microseconds.

The LTT patch is a powerful kernel-tracing system developed for the Linux kernel. LTT has been used for measuring the overhead of the Resource Reservations mechanism, but it is not strictly required in the middleware. The overhead introduced by LTT in our experiments is negligible with respect to the measured quantities. Indeed, the traced events are only the scheduling changes, which are few enough to keep in check the overhead introduced by the tracer.

Table I. Hardware-Software platform.

Hardware Platform	
Processor	AMD Athlon(tm) XP 2000
Frequency	1666MHz
RAM size	512 Mb
RAM disk partition size	128Mb
Software Platform	
Linux distribution	Fedora core 2
Compiler version	gcc v. 3.3.2
Kernel reference version	2.4.27
FFMPEG library version	0.4.8

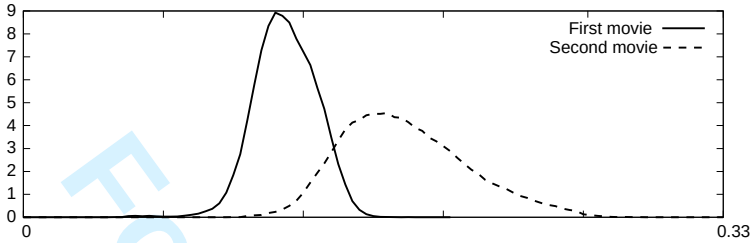
6.1. Performance evaluation

Before illustrating the experimental results of this section, it is useful to briefly introduce the metrics that we adopted to measure performance and the main factors that influence it.

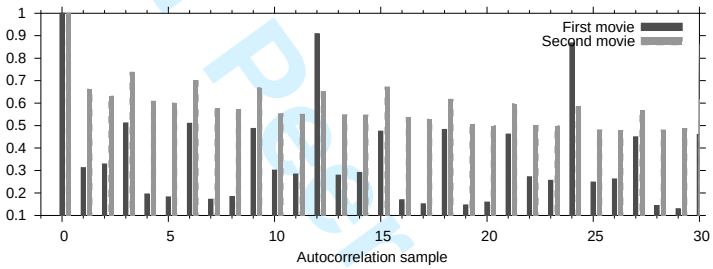
As discussed earlier, the design goal of a task controller is to maintain the system state in an equilibrium condition (in which ϵ_k is contained in the $[-e, E]$ set), and to restore the equilibrium in case of perturbations. Hence, a first important performance metric for our system is the experimental probability of ϵ_k falling into the target set $[-e, E]$. This metric, along with the average number of steps required to recover the equilibrium, quantifies the robustness of our design with respect to practical implementation issues. Moreover, a metric of interest is the average bandwidth allocated to the task: keeping this value as close as possible to the strict necessary allows us to maximise the use of the resource for applications having QoS guarantees (which is not doable using a fixed bandwidth allocation enriched with some reclaiming scheme [17]). Along with the quantitative performance metrics indicated above, a useful qualitative assessment can be made by a visual inspection of the experimental probability mass function (PMF), which should display small tails outside the target set $[-e, E]$.

The main factors degrading the target performance are: 1) prediction errors, 2) approximate knowledge of the application parameters required in Theorem 3 and Theorem 2, 3) overload situations. For the sake of simplicity, we will not consider here the third factor. This is equivalent to assuming that the considered task is of class A. The quality of the prediction can be evaluated considering the measure of the interval $[h_k, H_k]$ and the probability of $c_k \in [h_k, H_k]$. These parameters are influenced by the stochastic properties of process c_k and by the prediction algorithm.

To evaluate the impact of the c_k process, we considered two different video streams (see Figure 7.a). For both we used an adaptive reservation with period $P = 1ms$, which is $1/40^{th}$ of the task period.



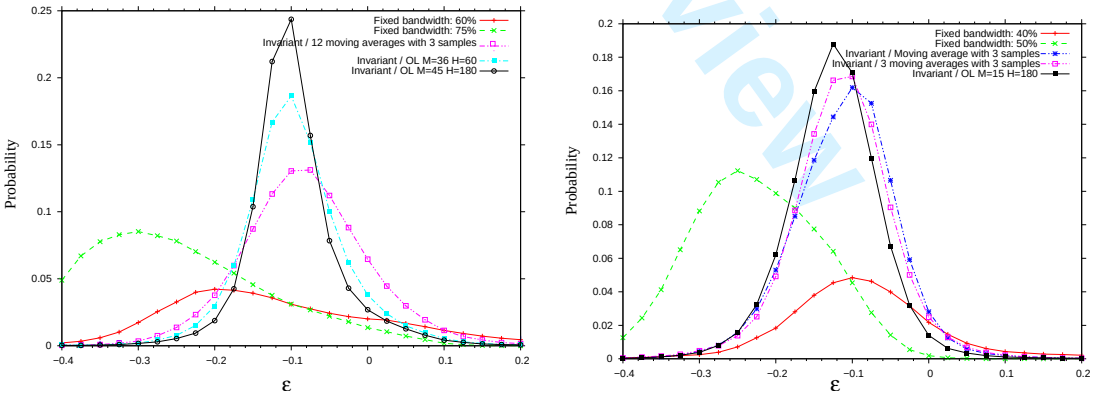
(a)



we also accounted for occational "spurious" spikes due to execution of long interrupt drivers. Indeed, in order to maintain an acceptable response time for Linux interrupts, we execute the drivers using the bandwidth of the interrupted tasks. In terms of our control design, this can be regarded as an unmodelled disturbance term. The specification for the task controller was to achieve a robust controlled invariant set of $e = 0.2T$ and $E = 0$. Another requirement was to recover from a maximum deviation of one period in at most $h = 10$ steps. We considered the applications of three types of predictors: MMA[3,12], OL[36,60] and OL[45,180]. A trial

Table II. Empirical statistics collected during the execution of the streams. Average μ_ϵ and variance σ_ϵ are expressed as percentage of the task period.

First video stream					
Predictor	p	μ_ϵ	σ_ϵ	μ_b	σ_b
Fixed bandwidth 20%	26.34%	2700%	5470%	-	-
Fixed bandwidth 25%	29.7%	-24.03%	12.1%	-	-
MMA[12,3]	76%	-6.87%	8.5%	20.41%	6.4%
OL[36,60]	86.61%	-8.31%	7.1%	20.64%	6.77%
OL[45,120]	89.93%	-8.49%	6.4%	20.68%	6.77%
Second video stream					
Predictor	p	μ_ϵ	σ_ϵ	μ_b	σ_b
Fixed bandwidth 13.3%	34.16%	6844%	98.89	-	-
Fixed bandwidth 17.8%	42.60%	-21.6%	0.089	-	-
MA[3]	92.75%	-9.73%	0.070	14.45%	4.8%
MMA[3,3]	93.18%	-10.04%	0.068	14.49%	4.8%
OL[15,180]	96.36%	-11.15%	0.063	14.67%	5.03%



(a) (b)

Figure 8. Comparison among the PMF achieved by various controllers and predictors on the first (a) and second (b) video stream

Table III. Measured execution time in CPU clock cycles, of the scheduling hooks.

	No load	10 Tasks	20 Tasks	30 Tasks
fork_hook_handler:	198	194	179	175
cleanup_hook_handler:	80	73	65	59
do_fork:	189920	229867	251982	271998
do_exit:	165154	160233	159187	160244
unblock_hook_handler:	526	645	701	805
block_hook_handler:	230	10719	13501	15274
sched_timeout:	336	420	460	520
schedule:	547	1612	2386	3205
qmgr_end_cycle:	1399	1414	1420	1426

The system workload is a relevant factor for both effects. Indeed, the management of the scheduling queues grows linearly with the number of tasks (in our implementation). Therefore, the execution of the hooks invoking the scheduler should display a similar behaviour.

The number of context switches introduced by the mechanism, for a given application, is also affected by the reservation period P and by the allocated bandwidth (which can be fixed or dynamic). The influence of P is evident. As far as the bandwidth is concerned, jobs with a higher bandwidth use a lower number of reservation periods to terminate, i.e., the number of context switches decreases.

We evaluated the overhead effects on the MPEG decoding application. The first set of experiments was aimed at assessing the execution time of the different hooks. We measured the duration of each hook for different workload conditions. In particular, we considered the execution of the task in isolation, and together with 10, 20 and 30 dummy tasks. Each dummy task consisted of an infinite loop and it was run by a resource reservation with fixed bandwidth and with reservation period varying in a range from $20ms$ to $40ms$. For each load condition, we ran the periodic task with a different bandwidth allocation. In particular we used a fixed value for the bandwidth of $1.1B_{av\mathbf{1}}$ (i.e.,10% greater than the average required bandwidth) and a feedback scheduler. For the latter, we used the same settings as above for the target set and a MA[3] predictor. The measurements were taken throughout the execution of 8000 frames of the stream, and we recorded the mean value. The results are reported in Table III. Clearly, to get the total execution time, we have to sum up the execution time of the hook to the one of the Linux standard function (see excerpt in Figure 4), which is reported in the table for the sake of completeness. Each number is expressed in clock cycles (on the considered system, 1000 clock cycles correspond to about 600 nanoseconds).

A first set of results displays the implementation cost for the Resource Reservation scheduler. These results are unaffected by the server period and by the bandwidth chosen for the task (both for what concerns its value and its allocation policy, i.e., static or adaptive). The execution time of the scheduling functions is approximately of the same order of magnitude as

that of the standard Linux scheduler. The `sched_timeout()` function is the handler of timer-related events: it is involved in budget handling (expiration and replenishment) and server scheduling. For this reason it is the main "responsible" of the context switches. As shown in the table, the growth rate of the average execution time of this function with respect to the workload is lower than the one of the standard Linux function (`schedule`).

We also evaluated the computation cost for the QoS Management algorithms (feedback and prediction), which is reported in the last row of the table (`qmgr_end_cycle`). As one would expect, in this case no real influence of the workload can be appreciated (only a slight increase due to cache effects introduced by other tasks in the system).

As far as server creation and termination is concerned, the additional overhead introduced by the `fork_hook_handler()` and `cleanup_hook_handler()` functions is negligible with respect to the corresponding Linux functions `do_fork()` and `do_exit()`.

In the second experiment, we evaluated the number of context switches and the total overhead introduced by our mechanisms. The overhead was measured as the ratio between the duration of the periodic task in different configurations using our middleware and the duration of the same task when executed in isolation without using any resource reservation mechanism. The measurements were taken varying the resource reservation period and the workload, and they are reported in Figure 9. Namely, for each workload situation, we varied the server period (on the horizontal axis). The first row is relative to a fixed bandwidth assignment, while the second raw to the use of feedback scheduling. The plots in the left column show the number of context switches and in the right column the total overhead. The dummy processes shared a total bandwidth equal to 10%. In all cases, we can see that the shapes of the plots reporting the number of context switches and the overhead are quite similar. This is because the function calls introducing most of the overhead are always associated to a context switch. The growth rate of both quantities is approximately linear with the number of dummy tasks.

The changes with respect to the reservation period follow approximately an hyperbolic pattern. This is perfectly consistent with our theoretical expectations. Indeed, if we think of a task in isolation and consider a periodic task with average execution time C, period P served by a reservation (Q, P), the average number of context switches in a time interval of length W is given by: $2\frac{W}{T}\lceil\frac{C}{Q}\rceil = 2\frac{W}{T}\lceil\frac{C}{BP}\rceil$, which can be approximated by an hyperbolic relation (the effect of ceiling can be neglected for low values of P). The presence of workload dummy tasks shifts upward this plot, but the qualitative appearance remains similar.

As a consequence, one would expect a lower overhead using greater bandwidth values, as it would decrease the number of context switches. This is highlighted in Figure 10, where the measurements have been taken in absence of dummy tasks. As it is possible to see, the overheads obtained for the adaptive reservation are intermediate between the ones obtained for low and large fixed values of the bandwidth.

As a final remark, the overhead introduced by our scheduling mechanism is, in our evaluation, acceptable: even for a reservation period of $2.5ms$, which is $1/16^{th}$ of the task period, our mechanism steals no more than 3% of the execution time of the MPEG player (in absence of dummy tasks).

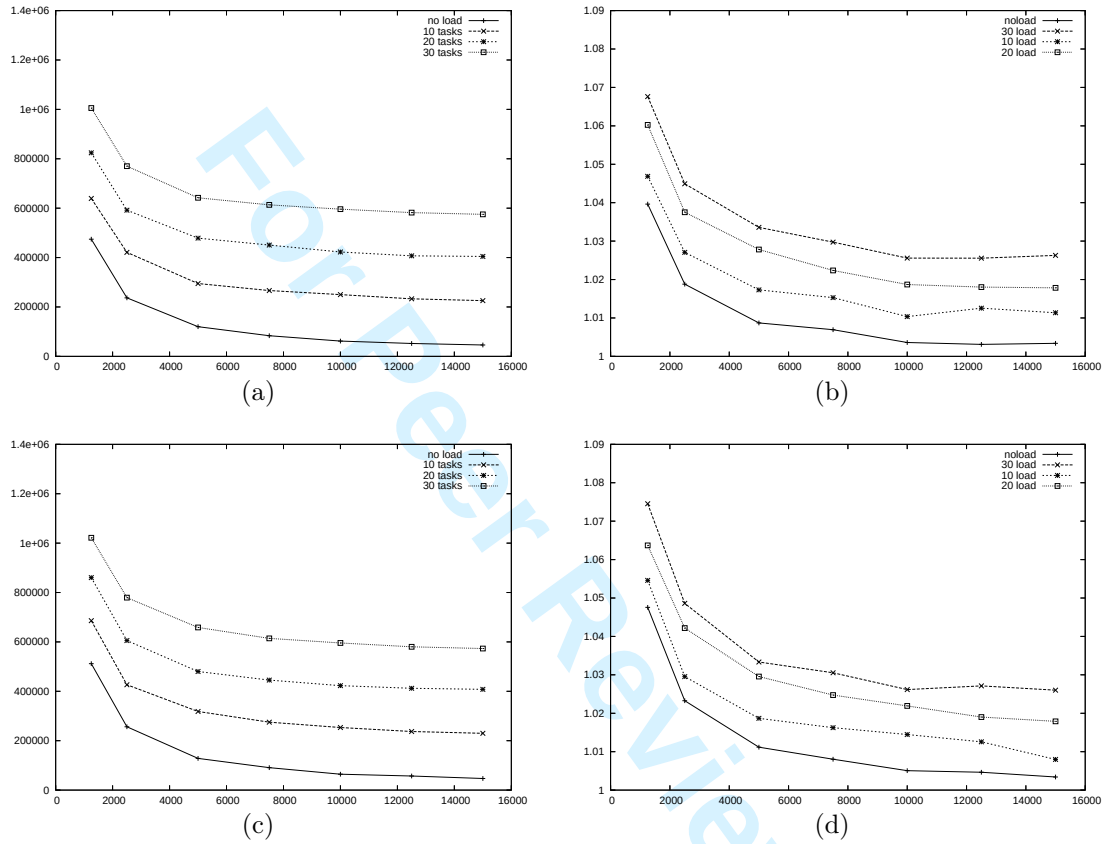


Figure 9. Number of context switches (left column) and relative overhead (right column) on the first video stream, with a fixed bandwidth of $1.1B_{av1}$ (a)(b), and with a dynamic allocation using an MA[3] predictor (c)(d). The horizontal axis represents the server period P.

7. Conclusions

In this paper we have shown a software infrastructure for applications that have to comply with soft real-time constraints. Our solution is based on a combination of the resource reservation scheduling algorithm and of a feedback based mechanism for dynamically adjusting the bandwidth. The adoption of the resource reservation algorithm allowed us to construct a precise mathematical model of the scheduler. Moreover, we stated in a sound mathematical framework the closed loop design goals (related to the system stability) and stated conditions and design solution allowing us to produce an effective control design. We offered a Linux based

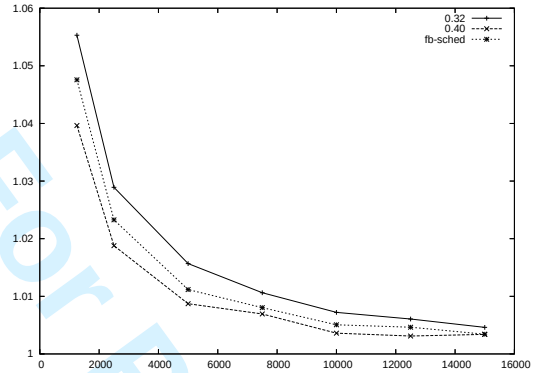


Figure 10. Comparison of the total relative overhead for fixed and dynamic bandwidth (no dummy tasks)

architectural solution that implements these ideas and that was designed to be: 1) flexible (it allows for the management of different types of resource and for the introduction of new control algorithms) 2) minimally invasive in terms of required kernel modifications, 3) efficient in terms of the introduced overhead. We also provided extensive experimental results that prove the effectiveness of the approach and its robustness to partial knowledge of the design parameters (collected through trial execution on segments of the application) and an evaluation of the introduced overhead.

This work can be extended in several directions. A first important possibility is to consider problems of coordinated allocation of multiple resources. As an example we could consider a pipeline of activities (decoupled by intermediate buffer) that use different types of resources (e.g., disk, network, CPU). We conjecture that the use of RR scheduler for different resources can represent an enabling paradigm for this type of technique. A theoretical analysis of this problem is under way. Moreover, we are planning the extension of the middleware to support the adaptive allocation of disk and network. Another important research issues is to coordinate scheduler level and application level adaptation. In particular, one of the possible action of the supervisor (in response to an overload condition) could be to require the application to lower its QoS level *via* a callback mechanism. To this aim, we need both a theoretical analysis of the problem and a viable architectural solution. Finally, from a purely technological point of view, we are studying solution for implementing adaptive reservation with a low complexity (e.g., using quantised priority levels and bit-mapped queues) and for using them with legacy applications (in a similar way as we currently do for tasks with fixed bandwidth).

REFERENCES

1. Adaptive resource allocation control for fair QoS management. *IEEE Transactions on Computers*, 56(3):344–357, 2007. Fumiko Harada and Toshimitsu Ushio and Yukikazu Nakamoto.
2. Sherif Abdelwahed, Nagarajan Kandasamy, and Sandeep 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.
3. L. Abeni, T. Cucinotta, G. Lipari, L. Marzario, and L. Palopoli. QoS management through adaptive reservations. *Real-Time Systems Journal*, 29(2-3):131–155, March 2005.
4. Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.
5. Luca Abeni and Giorgio Buttazzo. Adaptive bandwidth reservation for multimedia computing. In *Proceedings of the IEEE Real Time Computing Systems and Applications*, Hong Kong, December 1999.
6. Luca Abeni, Tommaso Cucinotta, Giuseppe Lipari, Luca Marzario, and Luigi Palopoli. Adaptive reservations in a linux based environment. In *Proceeding of the Real-Time Application Symposium (RTAS 04)*, Toronto (Canada), May 2004. IEEE.
7. Luca Abeni, Luigi Palopoli, Giuseppe Lipari, and Jonathan Walpole. Analysis of a reservation-based feedback scheduler. In *Proc. of the Real-Time Systems Symposium*, Austin, Texas, November 2002.
8. Adaptive quality of service architecture (AQuoSA), http://aquosa.sourceforge.net. Web site.
9. S.K. Baruah, N.K. Cohen, C.G. Plaxton, and D.A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 6, 1996.
10. Scott Brandt and Gary Nutt. Flexible soft real-time processing in middleware. *Real-time systems journal, Special issue on Flexible scheduling in real-time systems*, 22(1-2):77–118, January-March 2002.
11. Anton Cervin, Johan Eker, Bo Bernhardsson, and Karl-Erik Årzén. Feedback-feedforward scheduling of control tasks. *Real-Time Systems*, 2002.
12. 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.
13. Eric Eide, Tim Stack, John Regehr, and Jay 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.
14. Ffmpeg project, http://ffmpeg.sourceforge.net. Web site.
15. Sourav Ghosh, Jeffery Hansen, Ragunathan (Raj) Rajkumar, and John Lehoczky. Integrated resource management and scheduling with multi-resource constraints. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS04)*, pages 12–22, Washington, DC, USA, 2004. IEEE Computer Society.
16. Christopher D. Gill, Jeanna M. Gossett, David Corman, Joseph P. Loyall, Richard E. Schantz, Michael Atighetchi, and Douglas C. Schmidt. Integrated adaptive QoS management in middleware: A case study. *Real-Time Systems*, 29(2-3):101–130, march 2005.
17. G.Lipari and S.K. 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.
18. Ashvin Goel, Jonathan Walpole, and Molly Shor. Real-rate scheduling. In *Proceedings of Real-time and Embedded Technology and Applications Symposium*, page 434, 2004.
19. High resolution timers, http://high-res-timers.sourceforge.net. Web site.
20. K. Jeffay and S. M. Goddard. A theory of rate-based execution. In *Proceedings of the IEEE Real-Time Systems Symposium*, Phoenix, AZ, December 1999.
21. Yamuna Krishnamurthy, Vishal Kachroo, David A. Karr, Craig Rodrigues, Joseph P. Loyall, Richard E. Schantz, and Douglas C. Schmidt. Integration of QoS-enabled distributed object computing middleware for developing next-generation distributed application. In *LCTES/OM*, pages 230–237, 2001.
22. Abeni L. and Lipari G. Implementing resource reservations in linux. In *Real-Time Linux Workshop*, Boston (MA), USA, December 2002.
23. 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.
24. C. L. Liu and J. Layland. Scheduling alghorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1), 1973.
25. Linux trace toolkit, http://www.opersys.com/LTT. Web site.
26. C. Lu, J. Stankovic, G. Tao, and S. Son. Feedback control real-time scheduling: Framework, modeling and algorithms. *Special issue of RT Systems Journal on Control-Theoretic Approaches to Real-Time Computing*, 23(1/2), September 2002.

27. Luca Marzario, Giuseppe Lipari, Patricia Balbastre, and Alfons Crespo. IRIS: a new reclaiming algorithm for server-based real-time systems. In *Real-Time Application Symposium (RTAS 04)*, Toronto (Canada), May 2004.
28. Tatsuo 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.
29. 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.
30. Luigi Palopoli, Tommaso Cucinotta, and Antonio Bicchi. Quality of service control in soft real-time applications. In *Proc. of the IEEE 2003 conference on decision and control (CDC03)*, Maui, Hawai, USA, December 2003.
31. Luigi Palopoli, Tommaso Cucinotta, Giuseppe Lipari, and Luca Marzario. Adaptive management of QoS in open systems. Technical Report DIT-07-003, Department of Information and Communication technology, University of Trento, 2007.
32. Preemption patch, http://kpreempt.sourceforge.net. Web site.
33. Ragunathan Rajkumar, Chen Lee, John P. Lehoczky, and Daniel P. Siewiorek. Practical solutions for QoS-based resource allocation. In *RTSS*, pages 296–306, 1998.
34. Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi 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.
35. Dickson Reed and Robin Fairbairns (eds.). Nemesis, the kernel – overview, May 1997.
36. John Regehr and John 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.
37. Michael Roitzsch and Martin Pohlack. Principles for the prediction of video decoding times applied to mpeg-1/2 and mpeg-4 part 2 video. In *RTSS'06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 271–280, Washington, DC, USA, 2006. IEEE Computer Society.
38. Nishanth Shankaran, Xenofon D. Koutsoukos, Douglas C. Schmidt, Yuan Xue, and Chenyang Lu. Hierarchical control of multiple resources in distributed real-time and embedded systems. In *ECRTS'06: Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 151–160, Washington, DC, USA, 2006. IEEE Computer Society.
39. David Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the Third usenix-osdi*. pub-usenix, feb 1999.
40. Ian Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes E. Gehrke, and C. Greg Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1996.
41. Hideyuki Tokuda and Takuro Kitayama. Dynamic QoS control based on real-time threads. In *NOSSDAV'93: Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 114–123, London, UK, 1993. Springer-Verlag.
42. C.C. Wust, L. Steffens, R.J. Bril, and W.F.J. Verhaegh. QoS control strategies for high-quality video processing. In *Proceedings. 16th Euromicro Conference on Real-Time Systems - ECRTS 2004*, pages 3–12, 2004.
43. Ronghua Zhang, Chenyang Lu, Tarek F. Abdelzaher, and John 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.