UNIVERSITY
OF TRENTO

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

38050 Povo – Trento (Italy), Via Sommarive 14
http://www.dit.unitn.it

ADAPTIVE MANAGEMENT OF QOS IN OPEN SYSTEMS

Luigi Palopoli, Luca Abeni , Tommaso Cucinotta, Luca Marzario and Giuseppe Lipari

January 2007

Technical Report # DIT-07-003

Adaptive management of QoS in Open Systems

Luigi Palopoli, Luca Abeni , Tommaso Cucinotta, Luca Marzario, Giuseppe Lipari

Abstract

In this paper, we present a software architecture to support soft real-time applications, such as multimedia streaming and telecommunication systems, in open embedded systems. Examples of such systems are consumer electronic devices (as cellular phones, PDAs, etc.), as well as multimedia servers (video servers, VoIP servers, etc.) and telecommunication infrastructure devices. For such applications, it is important to keep under control the resource utilization of every task, otherwise the Quality of Service experienced by the users may be degraded.

Our proposal is to combine a resource reservation scheduler (that allows us to partition the CPU time in a reliable way) and a feedback based mechanism for dynamically adjusting the CPU fraction (bandwidth) allocated to a tasks. In particular, our controller enables specified Quality of Service (QoS) levels for the application while keeping the allocated bandwidth close to its actual needs. The adaptation mechanism consists of the combination of a prediction and of a feedback correction that operates locally on each task. The consistency of the system is preserved by a supervisor component that manages overload conditions and enacts security policies.

We implemented the framework in AQuOSA, 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 remarkably lower than the one introduced by other different solutions.

I. Introduction

Real-time system technology, traditionally developed for safety-critical systems, has been extended over the past years to support new application domains including multimedia systems, telecommunication networks, etc. Traditionally, such applications are referred to as soft real-time systems, as opposed to traditional hard real-time systems. Informally speaking, a soft real-time system is a system in which some temporal constraints can be violated without causing a system failure: however, the performance (or quality of service) of the system depends on the number of violated constraints and severity of such violations over a certain interval of time. One goal in executing a soft real-time application is to keep under control timing constraint violations.

The typical example of soft real-time system is a multimedia player (for example a DVD player or similar). One timing constraint for such application is the necessity to process video frames regularly according to the selected video frame rate. In case of a frame rate of 25 frames per second (fps), every 40 msec the application is expected to load data from disk, decode the frame, apply some filtering, and show the frame on the video screen. The application is usually built to be robust to timing violations. A delay in decoding a frame could not

L. Palopoli and L. Abeni are with the Università di Trento

T. Cucinotta, L. Marzario and G. Lipari are with the Scuola Superiore Sant'Anna, piazza Martiri della Libertà 33, 56127 Pisa, Italy

even be perceived by the users provided that the anomaly be kept in check (for instance, it can be compensated by using buffers). On the other hand, if the delays are too large, the application continues to work properly, but the user may perceive some bad effect on the screen. Therefore, the user *satisfaction* is inversely proportional to the number of timing violations. A measure of the user satisfaction is the so-called Quality of Service (QoS) of the application.

The term QoS is quite abused. In different research and application areas, the same term is used with different meanings, more or less precise. In all cases, the term is related to the impact that the system behavior (performance, etc.) has on the user satisfaction. In this paper, the notion of QoS we look at is tightly related to the real-time behaviour of the application.

The problem of providing QoS guarantees to time-sensitive applications is simply not there if we use a dedicated hardware for each application: we just have to use a powerful enough hardware! 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 user can concurrently access the videos, it is important to provide a guaranteed level of QoS to each user. Even on a user terminal (for example a desktop PC or a portable device) it is important to provide stable and guaranteed QoS to different applications.

In case of concurrent tasks, the problem arises of how to schedule 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 different possibility is to use hard real-time schedulers [27], [10], [28], either based on static priorities (e.g., Rate-Monotonic (RM)) or on dynamic priorities (e.g., Earliest Deadline First (EDF)). These approaches are very effective when: 1) the design goals are entirely focused on avoiding deadline violations, 2) an a priori knowledge of the worst case computation time is available. On the contrary, for the class of systems considered in this paper, we often do not have any such knowledge and we are interested in achieving a good trade-off between QoS (i.e. compliance with real-time constraints) and resource utilisation.

Therefore, a worst-case based allocation is not acceptable for applications (e.g., multimedia) with a wide variance in computation times, because the resource will be underallocated for most of the time. On the other hand, using standard priority-based hard real-time schedulers using average-case allocations is not supported by strong theoretical results and it suffers from a fundamental problem: it is hard to give *individual* execution guarantees to each task.

A class of real-time scheduling algorithm, referred as Resource Reservation algorithms, has been proposed in the literature to service soft real-time tasks. Such algorithms provide the fundamental property of *temporal protection* (also called *temporal isolation*): informally speaking, each task is *reserved* a fraction of the resource utilization, and its temporal behavior is not influenced by the presence of other tasks in the system. A more formal introduction to Resource Reservation algorithms will be made in Section II. Thanks to such algorithms, it is possible to allocate fractions of the resource (e.g. the CPU) to each task individually. By tuning the allocation of a task (for example based on measure of the average execution time), it is possible to have a first raw control

of the QoS provided by the task.

However, the availability of such scheduling mechanisms is not sufficient *per se* to ensure a correct temporal behavior to time-sensitive application. Indeed, the choice of the scheduling parameters that ensure a sufficient QoS level to the application can be difficult not only for the required knowledge of the computation time, but also for its timing variations. A fixed choice can lead to *local* degradations of the QoS (if it is strictly based on the average workload) or to a wasteful usage of resources. For this reason, we believe that the scheduling mechanism has to be complemented by a resource manager, which can operate on the scheduling parameters and/or on the application execution mode.

We distinguish the two main approach to adaptation as *resource level adaptation* and *application level adaptation*. The first kind of adaptation consists in dynamically changing the amount of resource reserved to a task depending on the task performance and QoS requirements, through an appropriate feedback control algorithm. The second kind of adaptation consists in adapting the application resource requirements, by changing the parameters of the application. For example, in a video player, it is possible to lower the resource requirements by lowering the frame rate. In this paper we address the first kind of adaptation and we do not explicitly deal with application level support for variable QoS. The issue is entirely orthogonal to our work and we are currently considering it for future inclusion into the framework.

The architecture presented in this paper aims at identifying an effective match between several contrasting needs:

- ensuring control in the QoS experienced by the application,
- maximising the degree of multiplexing in the use of shared resources,
- supporting the development of portable QoS sensitive applications by means of an abstract and easy-to-use API,
- offering an implementation based on widespread OS technologies (in our case the Linux environment) to take advantage of the vast legacy of existing applications and development tools,
- enabling the use of QoS services in a multiuser environment with appropriate protection against malicious or non-voluntary misuse.

A. State of the art

Different solutions have been proposed to schedule time-sensitive applications. Many of them are characterised by the *temporal protection property* (which will be formally defined in Section II). Scheduling algorithms guaranteeing temporal protection enable a controllable partitioning of the resource to the different applications. As each task receives a guaranteed fraction of a resource, it is preserved from the workload fluctuations of other tasks.

A first family of algorithms, including such proposals as the Proportional Share scheduling algorithm [43], [23], [22] and Pfair [8], revisits an idea initially developed in the domain of network packet scheduling [35], [15], [14], [45]: they approximate a *fluid flow* allocation of the resource (the Generalised Processor Sharing (GPS)), 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 the Resource Reservations schedulers [38], [4], [19], [39]. Since these mechanisms are the basis of our approach, we will formally introduce them in Section II.

The bandwidth allocated to each task is evidently related to the QoS that it provides. Therefore, orthogonal to the problem of guaranteeing a specified fraction of the CPU time (the scheduling mechanism) is the problem of making the appropriate choice in terms of the resource allocation *policy*.

We distinguish *application level* from *resource level adaptation* mechanisms. In the first case, the application can work in one of many *modes*, in each mode it requires a different resource utilisation and provides a different QoS. The application mode can be selected either off-line or on-line. For this kind of adaptation, Rajkumar et al [37] developed a framework called QRAM where the relationship between the QoS and the resource utilisation is specified by a continuous cost function. The QRAM design procedure allows one to associate several resources to the different applications and generates a vector of recommended resource utilisation. Defining the most appropriate cost functions may require a remarkable design effort. Moreover, the resulting allocation is static (i.e., each application receives a fixed fraction of resource).

An adaptive approach was proposed by Tokuda and Kitayama [44], in which a discrete number of QoS levels and corresponding resource utilisations is associated to each application. A *resource manager* chooses the right level for each application based on on-line QoS measurements. This idea has stemmed a large production of research results. For instance Wust et al. [46] proposed a controller based on an optimisation algorithm that solve a Markov decision problem for video-streams.

Brandt and Nutt [9] proposed a middleware layer of more general applicability, where a resource manager issues non-mandatory recommendations to the application concerning its QoS level. Other proposals performing adaptation in the middleware are [47] and [16], [18]. The latter evolve around the QuO [24] 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 general remark that applies to all approaches doing application-level adaptation is that they may require a substantial design and programming effort. Indeed, the application has to support several operation modes (in the case of discrete levels) and for each level the designer is required to quantify the resource requirements.

The *resource level adaptation* approach consists in dynamically adapting the allocation of resources to each task based on some feedback control scheme, and it is the approach we investigate in this paper. Roughly speaking, it works well when the system is not in permanent overload (i.e. the sum of the average requirements of all the tasks is below the resource capacity) and it is based on the property of statistical multiplexing of independent tasks: the probability that all tasks require their maximum execution time at the same instant is low, so we can take advantage of the space capacity of one task to give more time to another more demanding task.

A first proposal of this kind dates back to 1962 [13] and it applies to time-sharing schedulers. More recently, feedback control techniques have been applied to real-time scheduling [32], [40], [26], [30], [12], [11] and multimedia systems [42]. 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.

This problem can be overcome using such scheduling algorithms as Resource Reservations, which allow for a precise dynamic model of the system. This idea, which we call *Adaptive Reservations*, was pioneered in [5]. In [20] 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 [33]. The problem was further investigated in [34] and in [6], [2], where deterministic and stochastic nonlinear feedback control scheme taking advantage of the specific structure of the system model were 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 [1] and in [16], although in the latter paper most of the work is on the architectural side.

B. Contributions of this paper

This paper presents AQuOSA, a flexible, portable and lightweight software architecture for supporting Adaptive Reservations on top of a general-purpose Operating System. The architecture is well founded on formal scheduling analysis and control theoretical results. The architecture consists of a *scheduler*, a *control layer* and a *supervisor*.

From a scheduling point of view, a first important contribution of this paper is to identify under which conditions it is possible to change the bandwidth of the different tasks without compromising the properties of the Resource Reservations scheduler. Then, we propose a quantised model for the evolution of the QoS experienced by each task. In this way, we construct a dynamic model which is naturally dictated by the Resource Reservations scheduler (as compared to other works [2], [20] that refer to a fluid flow approximation of the system evolution). These results are presented in Section III.

Regarding the control layer, the adoption of a quantised model for the "plant" dynamics has a profound impact on control design, because control theory traditionally deals with continuous states, inputs and outputs. This problem is analysed in Section IV, where we propose a mathematically founded formulation of the control goals and a set of results and techniques to streamline the design of controllers operating with quantised measures for the system output. Our approach is based on a combination of controllers "locally" operating on each task (task controllers) and of a supervisor used to maintain the global consistency of the system.

The supervisor component is an innovative choice also from the perspective of our software infrastructure, where it is used to enforce security policies.

Another important contribution in our software architecture is the use of a mixed scheme in which part of the Adaptive Reservations mechanism is implemented in the kernel space (i.e. the scheduler), and part is

implemented in the middleware layer. To achieve maximum flexibility we allow the software modules of our architecture to run as kernel modules (for maximum efficiency) or as library function (for maximum portability and flexibility). Compared to other middleware layers [16], [18], the resulting architecture presents a good degree of flexibility with a limited overhead.

The improvement introduced by our ideas to the current industrial practise is, in our evaluation, evident. As of now, applications like Windows Media Player (TM), or Xine in the linux environment, are required to cope with bandwidth fluctuations by introducing large memory buffer and complex heuristics. This type of solution is expensive for embedded applications and it is not reliable and not portable. On the contrary, our approach offers guaranteed QoS levels and it makes for resource saving: as an example, we could save 65% of the buffer size using an adapted version of the Xine media player.

II. Resource Reservations

In this section, we formally introduce the problem of scheduling time-shared resources in a real-time system. Our solution is based on a Resource Reservations scheduler called Constant Bandwidth Server (CBS) [4]. For the sake of completeness and to help understand the dynamic model of the scheduler presented in Section III-C, we briefly recall in this section the concepts behind the Resource Reservations framework and the rules of the CBS algorithm.

A. 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 from the scheduler. After the task receives $c_j^{(i)}$ time units the job finishes at a *finishing time* labelled as $f_j^{(i)}$. We consider *preemptive* scheduling algorithms. In our model, job $J_j^{(i)}$ cannot receive execution units before $f_{j-1}^{(i)}$, 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: task $\tau^{(i)}$ 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: $r_j^{(i)} = d_{j-1}^{(i)} = jT^{(i)}$.

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 task* that can tolerate deadline misses. Therefore, a job is allowed to complete after the activation of the next job. In this case, activations are buffered: if a job $J_j^{(i)}$ is not able to complete before the next activation, the next job $J_{j+1}^{(i)}$ cannot execute until $f_j^{(i)}$. In other words, when $J_j^{(i)}$ finishes, if job $J_{j+1}^{(i)}$ has not yet arrived then $\tau^{(i)}$ blocks, otherwise $J_{j+1}^{(i)}$ is ready to execute.

B. The general idea of Resource Reservations

When executing multiple real-time applications in an *open* environment, where the workload parameters are scarcely knonw and highly variable in time, the notion of *temporal protection* plays a role of paramount importance. Restricting for the sake of simplicity to the case of independent tasks $\tau_1, \ldots, \tau_n$, a formal definition of this concept can be as follows:

Definition 1: A scheduling algorithm is said to guarantee temporal protection if the ability for each task $\tau^{(i)}$ to meet its timing constraints only depends on its task computation time and on its inter-arrival times, and not on the other tasks' workload.

In other words, tasks requiring too much execution time should not affect the performance of "correctly behaving" tasks. Classical real-time scheduling solutions such as Fixed Priority (FP) and Earliest Deadline First (EDF) [27] do not provide temporal protection.

A Resource Reservation $RSV^{(i)}$ for a task $\tau^{(i)}$ is described by a pair $(Q^{(i)}, P^{(i)})$, with the meaning that the task is *reserved* the resource for a maximum time $Q^{(i)}$ every $P^{(i)}$ units of time. $Q^{(i)}$ is the reservation *maximum budget* and $P^{(i)}$ is the reservation *period*. In general, task $\tau^{(i)}$ needs not to be periodic. Moreover, in case of periodic tasks, $P^{(i)}$ can be different from the task's period $T^{(i)}$. 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. A useful distinction [38] is between *hard* reservations and *soft* reservations. The former guarantees to task $\tau^{(i)}$ *exactly* $Q^{(i)}$ time units every $P^{(i)}$, even in presence of a *idle* processor On the contrary, if a *soft* reservation algorithm is used, a task $\tau^{(i)}$ is allowed to receive *more* than its reserved amount of resource $Q^{(i)}$ every $P^{(i)}$ if the processor is idle.

A reservation can be thought of, at a lower level, as a periodic hard real-time task with activation period $P^{(i)}$ and worst case execution time $Q^{(i)}$. Therefore, it is possible to use a classical scheduling algorithm like Rate-Monotonic (RM) or earliest deadline first (EDF) to schedule the reservations. To ensure the schedulability according to the selected scheduling algorithm (and hence the correctness of the Resource Reservations algorithm)), the standard Liu and Layland inequality [27] must be enforced:

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

with U^{lub} depending on the scheduling algorithm upon which the reservation system is implemented.

Among the different possibilities for implementing a Resource Reservations scheduler, we chose the Constant Bandwidth Server (CBS) [3], [4], which is briefly described in the next session.

C. The Constant Bandwidth Server

The CBS algorithm can be used to schedule periodic and aperiodic tasks. Moreover, a single *server* can be used to schedule either a single task or a multi-threaded application (in this case we get a hierarchy of schedulers). These possibilities are illustrated to their full extent in [4]. In this context, we recall the CBS rules restricting for simplicity to the case of single threaded applications and of periodic tasks.

In this case, each task $\tau^{(i)}$ is handled by a dedicated *server* $S^{(i)}$[1] implementing a reservation $RSV^{(i)} = (Q^{(i)}, P^{(i)})$. A server is a *schedulable entity* that maintains two internal variables, the current budget $q^{(i)}$ and the *scheduling deadline* $s^{(i)}$ (also known as server deadline). Initially, both $q^{(i)}$ and $s^{(i)}$ are initialised to 0.

[1]The name *server* is used for historical reasons. Resource reservation algorithms are the "evolution" of a class of scheduling algorithms, denoted as *aperiodic servers* used for handling soft aperiodic tasks in hard real-time systems.

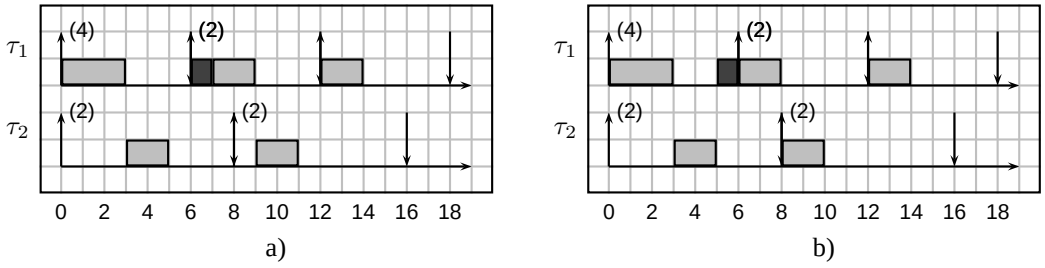

Fig. 1. Example of schedule produced by algorithm CBS with a) hard reservations, b) soft reservations.

The CBS algorithm is based on the EDF scheduling algorithm: among all active servers, the one with the earliest scheduling deadline is selected and the corresponding job is executed. By using EDF, the system can reach the maximum possible utilisation [27], thus $U^{lub} = 1$.

The CBS algorithm can be modified to implement either hard reservations or soft reservations. The hard version works as follows:

1) when a new job $J_j^{(i)}$ arrives at time t:
   - if a previous job of task $\tau^{(i)}$ is not yet finished, the activation is buffered;
   - if no jobs of task $\tau^{(i)}$ are ready for execution, the current budget and deadline are updated as follows: $q^{(i)} \leftarrow Q^{(i)}$ and $s^{(i)} \leftarrow t + P^{(i)}$.
2) when the task $\tau^{(i)}$ executes (i.e., when its server is scheduled by EDF), the current budget is decreased accordingly;
3) when a job completes: if there is a pending activation, the server remains active and continues to execute the new job;
4) if the current budget is exhausted and the job has not completed yet, the server is said to be *depleted*, and the served task cannot execute until the server budget $q^{(i)}$ is *replenished*, which happens at the current scheduling deadline $s^{(i)}$;
5) at the replenishment time, the budget of a depleted server is recharged to its maximum value, and the server deadline is increased by $P^{(i)}$: $q^{(i)} \leftarrow Q^{(i)}$ and $s^{(i)} \leftarrow s^{(i)} + P^{(i)}$.

In Figure 1.a, we show an example of schedule produced by the CBS algorithm with hard reservations. In this example, we show two periodic tasks: $\tau^{(1)}$ with a period of $T^{(1)} = 6$ and with variable execution time; and $\tau^{(2)}$ with period $T^{(2)} = 8$ and constant execution time equal to $c_j^{(2)} = 2, \forall j$. In the figure, the execution of each task is shown on a separate horizontal line, and the upward arrows denote the arrival times of the jobs. The numbers in parenthesis near the upward arrows denote the execution requirements of the jobs. The downward arrows denote the deadlines of the jobs. We assume that task $\tau^{(1)}$ is assigned a server $S^{(1)}$ with $Q^{(1)} = 3$ and $P^{(1)} = 6$, and task $\tau^{(2)}$ is assigned a server $S^{(2)}$ with $Q^{(2)} = 2$ and period $P^{(2)} = 8$. Notice that $B^{(1)} + B^{(2)} = \frac{3}{4} < 1$.

At the beginning, server $S^{(1)}$ has a deadline $s^{(1)} = 6$ and it is the earliest deadline server, so task τ_1 is executed. However, the execution requirement of $J_1^{(1)}$ is 4 and the server budget is 3; hence, at time 3 the task is suspended until time 6 (rule (4)). At time 6, the server budget is recharged (rule (5)). Meantime, $J_2^{(1)}$

has arrived at time 6; this second job has to wait for the first job to complete, which happens at time 7. In Figure 1.a, we show the late execution of job $J_1^{(1)}$ with a darker box. Notice that at time 5 the processor is idle, because the first server is suspended, and the second server has completed its job. The algorithm is clearly *non-work-conserving*.

In the soft reservation version, Rule 5 is eliminated while Rule 4 is modified as follows:

(4) if the current budget is exhausted and the job has not yet completed, the current budget is immediately recharged to $Q^{(i)}$ ($q^{(i)} \leftarrow Q^{(i)}$) and the scheduling deadline is postponed to $s^{(i)} \leftarrow s^{(i)} + P^{(i)}$. Therefore, the EDF ready queue has to be re-ordered and a preemption may occur.

In Figure 1.b we show the schedule produced by the soft reservation version of the CBS algorithm. The main difference is at time 3: the server $S^{(1)}$ is not suspended, but its deadline $s^{(1)}$ is postponed to 12 and its budget $q^{(1)}$ is immediately recharged. As a consequence, at time 5, server $S^{(1)}$ can execute and complete the execution of $J_1^{(1)}$. Notice that the server budget $q^{(1)}$ is decreased anyway, so at time 6 the remaining server budget is $q^{(1)} = 2$. We denote by the symbol $s_j^{(i)}$ *the last* deadline assigned for the j-th job.

The CBS algorithm satisfies the properties stated in the following results.

Theorem 1 (Temporal Protection [4]): If Equation (1) holds, then at any time t, and for every active server $S^{(i)}$, $t \leq s_j^{(i)}$. In other words, each server always executes its assigned budget before its current scheduling deadline. In the hard reservation version, it also holds $t \geq s^{(i)} - P^{(i)}$, i.e. the server always executes the assigned budget $Q_i^{(i)}$ inside an interval $s^{(i)} \geq t \geq s_j^{(i)} - P^{(i)}$.

Corollary 1: Given a task $\tau^{(i)}$, handled by a server $S^{(i)}$, suppose job $J_j^{(i)}$ finishes at $f_j^{(i)}$ and the server scheduling deadline at time $f_j^{(i)}$ is $s_j^{(i)}$. Then:

1) in the soft reservation version $f_j^{(i)} \leq s_j^{(i)}$;
2) 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.

III. Adaptive reservations

Adaptive Reservations are an extension of the Resource Reservations technology that copes with the following problem: how should the $(Q^{(i)}, P^{(i)})$ pair be dimensioned in presence of scarcely known and/or time-varying execution times. To this regard, a static partitioning of the CPU time (based on fixed choice for $(Q^{(i)}, P^{(i)})$) may lead to infeasible or inflexible choices. To address this problem, 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.

More specifically, such a feedback mechanism can be constructed based on:

- a *measured value*, used as input to the feedback mechanism;
- a *dynamic model* of a reservation, which can be used to design the feedback control strategy;
- an *actuator*, which permits to apply the feedback action to change the system behaviour.

In the rest of this section we will describe each of the above items. The model of Adaptive Reservations described in this section is elaborated from the one presented in [7].

Hereinafter, we assume the use of a hard reservation CBS for the model and for control design. Most results can be reformulated, in a weaker form, also for soft reservations, but we do not do it in this paper for the sake of brevity.

A. Measured value: scheduling error

The ideal goal of an adaptive reservation is to schedule $\tau^{(i)}$ so that $\forall j, f_j^{(i)} = d_j^{(i)}$. In this way, not only it is guaranteed that the task progresses respecting its timing constraint but also the quantity of CPU alloted to the task is exactly the one it needs. Therefore, the deviation, $f_j^{(i)} - d_j^{(i)}$ is a reasonable choice as a measured value. 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*. However, in view of Corollary 1, we are provided with the upper bound $s_{(j)}^{(i)}$ and, since we use hard reservations, with the lower bound ($s_{(j)}^{(i)} - P^{(i)}$). Therefore, if we are able to control the quantity $\epsilon_j^{(i)} = s_{j-1}^{(i)} - d_{j-1}^{(i)}$, we get reasonably close to our ideal objective (this quantity can be regarded as a quantised measure of $f_{j-1}^{(i)} - d_{j-1}^{(i)}$).

B. The dynamic model of a reservation

A considerable advantage of the choice of the measured value 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. The discussion below will be referred to a single task. Hence for notational convenience, we will drop the task index in this section, and wherever convenient in the rest of the paper.

In order to construct such a model, we have to consider two cases: 1) $s_k \leq r_k + T$, 2) $s_k > r_k + T$.

To clarify the meaning of the variables in the two cases, consider the example shown in Figure 2. Here we have a task τ with a period T that is assigned a server S with a period $P = T/2$. The first case is shown in Figure 2.a. Job J_1 uses only 2 reservation periods and finishes before the end of its period. In this case, the subsequent job (J_2) becomes active right after its arrival and the number of server periods necessary to terminate the job is $\left\lceil \frac{c_2}{Q_2} \right\rceil$. Hence the last server deadline for job J_2 is given by: $s_2 = r_2 + \left\lceil \frac{c_2}{Q_2} \right\rceil P = d_1 + 2P$ and the scheduling error $\epsilon_3 = s_2 - d_2 = d_1 + \left\lceil \frac{c_2}{Q_2} \right\rceil P - d_2 = 2P - T = 0$.

The case $s_k > r_k + T$ is more involved (see Figure 2.b). In this case jobs J_{k-1} and J_k share a reservation period. During this shared period the system evolves with the previously assigned budget Q_{k-1}. To express this dependency, and write the dynamic equations of our system, it is useful to introduce another state variable, called x_k (defined in the range $[0, Q[$, that represents the amount of time used by J_k in its first reservation shared with J_{k-1}.

In the example in Figure 2.b, J_1 uses 3 reservation periods and the residual budget available for J_2 in its first period is $x_2 = 1$. Being $c_2 > x_2$ the number of reservation periods (in addition to the shared reservation period) needed by job J_2 to terminate is given by $\left\lceil \frac{c_2 - x_2}{Q_2} \right\rceil$. The resulting server deadline is given by $s_2 = s_1 + \left\lceil \frac{c_2 - x_2}{Q_2} \right\rceil P$.

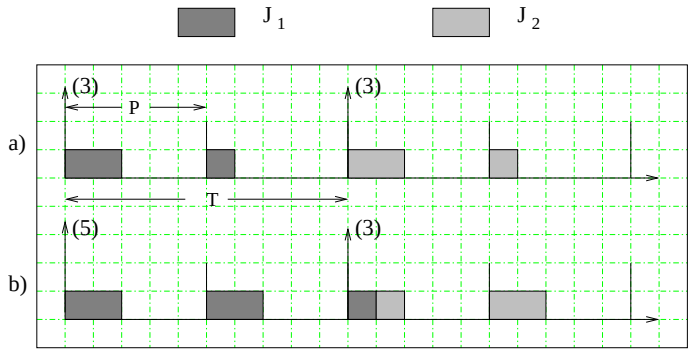

Fig. 2. Internal state x_j. In case a), the first job finishes before the end of its period, hence $x_2 = 0$; in case b), the first job consumes 3 reservation periods, and consumes 1 capacity unit in the last reservation period, hence $x_2 = 1$.

Generally speaking, it is possible to write the following discrete event model:

$$\begin{aligned} s_0 &= \left\lceil \frac{c_0}{Q_0} \right\rceil P \\ s_k &= \begin{cases} d_{k-1} + \left\lceil \frac{c_k}{Q_k} \right\rceil P & \text{if } s_{k-1} \leq d_{k-1} \\ s_{k-1} & \text{if } s_{k-1} > d_{k-1} \text{ and } c_k \leq x_k \\ s_{k-1} + \left\lceil \frac{c_k - x_k}{Q_k} \right\rceil P & \text{if } s_{k-1} > d_{k-1} \text{ and } c_k > x_k \end{cases} \end{aligned} \tag{2}$$

and

$$\begin{aligned} x_0 &= 0 \\ x_{k+1} &= \begin{cases} 0 & \text{if } s_{k-1} \leq d_{k-1} \\ x_k - c_k & \text{if } s_{k-1} > d_{k-1} \text{ and } c_k \leq x_k \\ Q_k \left\lceil \frac{c_k - x_k}{Q_k} \right\rceil - c_k & \text{if } s_{k-1} > d_{k-1} \text{ and } c_k > x_k. \end{cases} \end{aligned} \tag{3}$$

Considering the definition of scheduling error, from Equation (2), it is possible to write:

$$\epsilon_{k+1} = s_k - d_k = \begin{cases} \left\lceil \frac{c_k}{Q_k} \right\rceil P - (d_k - d_{k-1}) = \left\lceil \frac{c_k}{Q_k} \right\rceil P - T & \text{if } \epsilon_k \leq 0 \\ s_{k-1} - d_{k-1} - (d_k - d_{k-1}) = \epsilon_k - T & \text{if } \epsilon_k > 0 \text{ and } c_k \leq x_k \\ \epsilon_k + \left\lceil \frac{c_k - x_k}{Q_k} \right\rceil P - T & \text{if } \epsilon_k > 0 \text{ and } c_k > x_k \end{cases} \tag{4}$$

The dynamic model of a reservation is described by the pair of Equations 3 and 4. However, we are interested in controlling only the evolution of ϵ_k (our interest in x_k is very limited). Therefore, observing that x_k is an exogenous, bounded ($x_k < Q_{k-1}$) and measurable term, we can write the following simplified model:

$$\epsilon_{k+1} = \begin{cases} \left\lceil \frac{c'_k}{Q_k} \right\rceil P - T & \text{if } \epsilon_k \leq 0 \\ \epsilon_k + \left\lceil \frac{c'_k}{Q_k} \right\rceil P - T & \text{if } \epsilon_k > 0 \end{cases} \tag{5}$$

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

$$c'_k = \begin{cases} c_k & \text{if } \epsilon_k \leq 0 \\ 0 & \text{if } c_k \leq x_k \\ c_k - x_k & \text{otherwise.} \end{cases}$$

If x_k were not measurable, we could replace c'_k in Equation (5) with c_k; the resulting model would represent the evolution of a quantity which is an upper bound of the scheduling error. Indeed, it is exactly the evolution that we would get if the residual computation time x_k in a reservation shared between two jobs were not utilised.

The model in Equation (5) is the one that we shall use in our control design. In this model, $\varepsilon_k^{(i)}$ plays the role of state variable, $c^{(i)}$ is an exogenous disturbance and $Q_k^{(i)}$ can be used as a command variable to steer the evolution of the system (we will later discuss some constraints in utilising this actuator).

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 $c^{(i)}$ to the machine's clock cycles.

C. The actuation mechanism

In our feedback loop, we chose to use the maximum budget Q_k as an actuator (keeping constant reservation period P). In this section we discuss when and how changes in Q_k do not compromise the consistency of the Resource Reservations scheduler.

In the following, for the sake of simplicity, we assume that the budget of the server can be changed only at the beginning of a server period (in principle it could be possible to do otherwise *via* tedious and complex computations).

We define a global variable $B^{tot}(t)$ that keeps track of the total system bandwidth at all instants. In order to guarantee schedulability of the reservations, the following cases must be considered:

- Suppose that, at time t, a server needs to decrease its budget from Q to Q' (and its bandwidth from B to B'). The new budget will be applied in the next server instance; at the end of server period, the total bandwidth can be decreased to
$$B^{tot}(t)_{new} = B^{tot}(t) - B + B'. \tag{6}$$
- Suppose that, at time t, a server needs to increase its budget from Q to Q' (and its bandwidth from B to B'). We must consider two possible conditions:
  1) if $B^{tot}(t) - B + B' \leq U^{lub}$, then the total bandwidth is immediately increased as in Equation (6), while the server budget will be increased from the next server instance;
  2) if $B^{tot}(t) - B + B' > U^{lub}$, then this request cannot be accepted, because it would jeopardise the consistency condition of the CBS. It is then possible to operate with different policies. One possibility is to *saturate* to $B' = U^{lub} - B^{tot}(t) + B$ and we fall back in the previous case. If, on the contrary, we want to grant the full request, we have to reduce the bandwidth of other servers. Therefore, the server must first wait for the other servers to decrease their bandwidth requirements so that the new required bandwidth can be accommodated. Therefore, the actuation is delayed until the total bandwidth reaches a value
$$B^{tot}(t) \leq U^{lub} - B + B'.$$

In any case, when this situation occurs, it is clear that the actuation may be delayed and its value could be saturated if the requests from the sever exceed the total system bandwidth. In Section IV-D we will describe the *supervisor* module, which implements different policies that can be used to manage this situation.

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.

IV. The feedback control mechanism

A. 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).

Therefore, our first goal is to formally define a notion of "desired" equilibrium that befits our model. Considering a single task controller, as discussed in the previous section, an *ideal allocation* is for us one that makes $\varepsilon_k = 0, \forall k$. However, this ideal allocation is not implementable since it entails $Q_k = \frac{c_k P}{T}$ at the equilibrium, which presumes a predictive knowledge of the computation time of the next job. An approximation of the ideal allocation can be achieved assuming some - albeit uncertain - knowledge of the next computation time. In our particular case, the controller relies on the estimation of an interval $\mathcal{C}_k = [h_k, H_k]$, such that $c_k \in \mathcal{C}_k$. Due to this imprecise knowledge, the equilibrium point is stretched to a set in which the controller is able to keep the system state, contrasting the action of the disturbance term. 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 denotes the command variable whose values are restricted to a set $\mathcal{Q}$, and c_k is an exogenous disturbance term belonging to a closed and bounded set $\mathcal{C}_k \subseteq \mathcal{C}$. A set $\mathcal{I} \subseteq \mathcal{E}$ is said a *robustly controlled invariant set* (RCIS), if there exists a control law $g : \mathcal{E} \times \{\mathcal{C}\} \rightarrow \mathcal{Q}$, $q_k = g(\varepsilon_k, \mathcal{C}_k)$, 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 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. In particular, the level of QoS experienced by the task is governed by E. By picking large values of E, we allow for considerable delays in the termination of each job. On the other hand, parameter e is related to the potential excess of bandwidth received by the task. In particular a large value for e, allows the task to receive much more than the minimum it requires ($\frac{c_k}{T}$). 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 we state the following definition.

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 $\mathcal{J}$ in h steps, if there exists a control law $g : \mathcal{E} \times \{\mathcal{C}\} \rightarrow \mathcal{Q}$, $q_j = g(\varepsilon_j, \mathcal{C}_j)$, such that $\forall \varepsilon_k \in \mathcal{J} \backslash \mathcal{I} : \exists \overline{h} \leq h$ such that $c_j \in \mathcal{C}_j, \forall j = k, \ldots, k + h \rightarrow \varepsilon_{k+\overline{h}} \in \mathcal{I}$
2) $\mathcal{I}$ is *reachable* from $\mathcal{J}$ if there exist h such that $\mathcal{I}$ is h-step-reachable from $\mathcal{J}$.

In some cases, we can require that the target set $\mathcal{I}$ in the reachability definitions above is the RCIS, chosen as equilibrium for the system. The set $\mathcal{J}$ is an interval that somehow defines the maximum deviation that a task is allowed to take from its desired condition (i.e., the maximum delay that it makes sense for a task to accumulate). If $\mathcal{J}$ is chosen equal to the entire state space $\mathcal{E}$, 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
- **Class C**: no guarantee is offered but the controller trys 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 piecewise constant c_k).

B. The 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 the 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 the control scheme in Figure 3, which is referred to as *decentralised control* in the control literature. Each task is attached a dedicated controller, called *task controller*, that uses only the information collected from the task. Roughly speaking, a task controller is responsible for maintaining the QoS provided by the task in specified bounds with the minimum impact on CPU utilisation. Still, the total bandwidth request from the different task controllers is allowed to exceed the bound in Equation (1). To handle this situation (henceforth referred to as *overload*), a component called *supervisor* is used to reset the bandwidth allocated to the different tasks within the appropriate levels.

One final remark is on the computation workload of the control component, which is to be kept under very low bounds (the system has a profound interaction with the scheduler, which operates in time frames of a few micro-seconds). This constraint rules out such design approaches as dynamic programming or model predictive control, which feature excellent performance at the price of unacceptable computation resources.

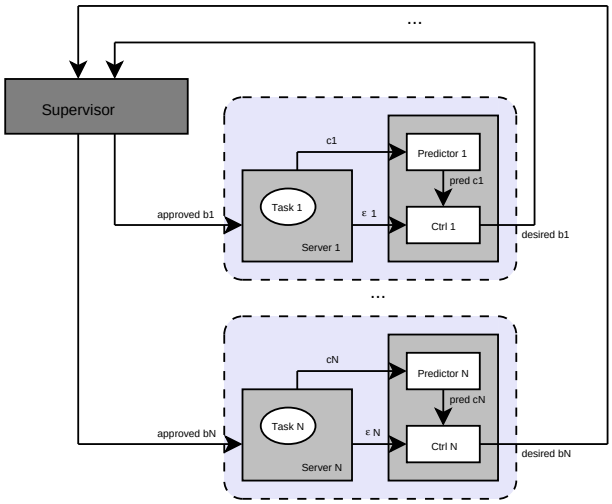

Fig. 3. The control scheme: each task is attached a local feedback controller and the total bandwidth requests are mediated by a supervisor.

C. Task controllers

A task controller is comprised of two components: a feedback controller and a predictor. The sensors located inside the RSV scheduler return at the termination of each job the computation time of the job just concluded and the experienced scheduling error. The former information is used by the predictor to estimate a range $\mathcal{C}_k$ containing the computation time of the next job: $c_k \in \mathcal{C}_k$. The feedback controller tries to fulfil the design goals of the task deciding the budget Q_k; to this end it relies on the measurement of ε_k and on the estimated range $\mathcal{C}_k$.

1) Feedback controller: In the following, we will assume that the feedback controller operates with perfect predictions (i.e., $c_k \in \mathcal{C}_k$). 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 The first important point to tackle is how to attain robust controlled invariance (for tasks of class A). Since ε_k evolves in the lattice $\mathcal{E}$ introduced in Section IV-B, the RCIS 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 simplicity, we will restrict the analysis to the case of RCIS sets slightly smaller than the task period, precisely satisfying $\hat{e} + \hat{E} < L - 1$. In particular, we consider the two following issues: 1) which choices of $\hat{e}$ and $\hat{E}$ yield an attainable RCIS for a given $\mathcal{C}_k = [h_k, H_k]$ and for a given saturation value $\overline{Q}$, 2) which feedback control laws enact robust control invariance of $\mathcal{I}$. Both points are addressed in the following.

Theorem 2: Consider system in Equation 5 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 $\alpha_k \triangleq \frac{h_k}{H_k}$. The following statements hold:

1) $\mathcal{I}$ is a RCIS if and only if

$$\hat{e} + \alpha_k \hat{E} > L(1 - \alpha) - 1 \wedge \overline{Q} > \frac{H}{L}, \tag{7}$$

where $\alpha \triangleq \inf_k\{\alpha_k\}$, $H \triangleq \sup_k\{H_k\}$

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

$$Q_k \in \left[\frac{H_k}{L + \hat{E} - \frac{S(\varepsilon_k)}{P}}, \min\left\{ \frac{h_k}{L - 1 - \hat{e} - \frac{S(\varepsilon_k)}{P}}, \overline{Q} \right\} \right[\tag{8}$$

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

Proof: See appendix. ■

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 8 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 violation of 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.

Reachability The following result shows how to steer the system from an arbitrary initial state back into an interval. We consider reachability of a set of the form $\mathcal{I} = [-(L-1)P, \hat{E}P]$, with $\hat{E} \in \mathbb{N}$ from a set of the form $\mathcal{J} = [-(L-1)P, \Gamma P]$, with $\Gamma > \hat{E}$. In case of a class A task, we have to steer the system into a RCIS. In view of Equation (5), control actions to reach $\mathcal{I}$ starting from a negative scheduling error are no different from those necessary to preserve invariance of $\mathcal{I}$. In other words, we can switch to the control law necessary to enact control invariance as soon as ε_k becomes less than or equal to E. Therefore, bounding the lower extremal point of the target set to the minimum possible value for the scheduling error ($-(L-1)P$) is not a loss of generality.

Theorem 3: Consider the system defined by Equation (5). Consider the intervals $\mathcal{I}$ and $\mathcal{J}$ as defined above. Let $\tilde{H}_h = \sup_k \frac{1}{h} \sum_{j=k}^{k+h-1} \left\lceil \frac{H_j}{\overline{Q}} \right\rceil \overline{Q}$ and assume that limit $\tilde{H} \triangleq lim_{K \rightarrow +\infty} \frac{1}{K} \sum_{j=0}^{K-1} \left\lceil \frac{H_j}{\overline{Q}} \right\rceil \overline{Q}$ exist and be finite. The following statements hold true:

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 $\overline{Q} \geq \frac{h\tilde{H}_h}{hL - \Gamma + \hat{E}}$;

Proof: See appendix. ■

Also to demonstrate this theorem, we used a constructive proof that produces a *minimum time* strategy for attaining reachability, i.e., one that uses the entire amount of available bandwidth to drive the system back

into the equilibrium as soon as possible. Provided that necessary conditions are met, it is also possible to use exponential reduction strategies that delay the restoration of the equilibrium but, in the mean time, diminish the amount of used bandwidth.

Control to zero. Assume that the system evolves in a RCIS, using a control law compliant with Equation 8. The third ancillary requirement for tasks of class A can be rephrased as follows: if at some instant k the controller had a hint $\tilde{c}_k$ on the exact value of the next computation time (as it might happen if the input features a piecewise constant behaviour), is the system able to reduce the error to 0 in one step ? This possibility is shown in the following

Fact 1: Assume that the system evolves in a RCIS $\mathcal{I} = [-e, E]$ using a control law compliant with Equation 8. 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 8 which controls the scheduling error to zero in one step $\forall \varepsilon_k \in \mathcal{I}$, is given by:

$$\mathcal{Z} = \left\{ \tilde{c} \mid H\frac{T-E-P}{T} < \tilde{c} \leq h\frac{T}{T-P-e} \wedge \tilde{c} \leq \frac{\overline{Q}}{P}(T-E) \right\}, \tag{9}$$

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 8.

Proof: See appendix. ■

2) The predictor component: As shown in Theorem 2, the accuracy of the prediction improves the steady state performance of the system. Indeed, a tight prediction interval enables a small RCIS. On the other hand, even though our scheme is resilient to occasional prediction failures, it is important that the prediction interval $[h_k, H_k]$ be correct with a good probability (see Remark 1). 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. Hence, it is presumable that the best performance (e.g., the tightest RCIS) can only be attained based on a strong knowledge on the application. For this reason, as shown below, the application programmer is left with the possibility of linking his/her own predictor to the task controller. Nonetheless, we have found it useful to provide a set of simple "library" predictors, displaying a fair level of performance for a wide set of applications. Our main concern was, in this case, to keep the prediction's complexity under very low bounds.

For the sake of simplicity, we restricted our attention to FIR structures, i.e. the predicted value $\hat{c}_k$ is given by:

$$\hat{c}_k = \sum_{j=1}^{M} w_j c_{k-j}. \tag{10}$$

Coefficients w_j are called "taps", in the linear filter literature. If all taps are chosen equal and normalised to the value $\frac{1}{M}$, we get a moving average, which is an estimator for the mean value μ_k of the process. Likewise, it is possible to estimate the mean squared value, and hence the standard deviation σ_k. Thereby, the predicted interval can be estimated as $h_k = \mu_k - \alpha\sigma_k$, and $H_k = \mu_k + \alpha\sigma_k$, where $\alpha > 0$ is a constant that permits to trade prediction accuracy against probability of a wrong estimation. Both μ_k and σ_k can be obtained with a

fixed computation time and a linear memory occupation (with respect to the horizon M). This type of predictor will be henceforth denoted as MA[n] (meaning Moving Average of length n).

There are some cases (e.g. in MPEG decoding), in which the application has a periodic pattern, which is reflected in the autocorrelation structure of the process. A small departure from the simple idea sketched above is, in this case, to use more than one moving average; for instance as many moving averages as the periodicity of the process. If S is the period of the sequence, the spatial complexity of the algorithm grows linearly with the product MS. We denote this type of predictor as MMA[m,S], meaning that it runs S moving averages of length m.

The algorithms proposed above are very efficient and offer an acceptable performance (as shown in the experimental section below). In the general case, the taps w_j need not be equal (since they reflect the correlation structure of the process). A natural choice is to choose them as a result of an optimisation program (e.g. least square). 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 apply the least square optimisation algorithm and - henceforth - proceed with the computed taps. We will denote this solution as OL[m,n], where m denotes the number of taps and n the length of the training set.

For time-varying processes, it is possible to have a "continuous training": the taps w_j become time varying in their turn ($w_{j,k}$), and a recursive optimisation algorithm is executed on-line to update the taps along with the prediction. To this regard, it is possible to use a recursive formulation of the least square algorithm (RLS), or more simply a gradient descent algorithm. This problem has been widely explored by researchers active in the field of adaptive filtering [41]. A solution of this kind would lead to an unsustainable overhead in our architecture, thus is has not been considered in the results exposed in this paper.

D. The role of the Supervisor

The main role of the supervisor is to ensure that each task receives its guaranteed amount of bandwidth. This is relatively easy when the system is not overloaded. In this case, in principle, the supervisor can be transparent and simply propagate the bandwidth requests from the task controllers to the scheduler. In fact, even in this situation (as shown later), a task controller might issue bandwidth requests that violate security rules (unrelated to the feedback mechanism), thus inducing a supervisor correction. The situation in which the supervisor plays the most important role is when the system is overloaded. In this case, the sum of the bandwidth $B^{(i)}$ required by the tasks exceeds the U_{lub} bound and the supervisor cannot grant the whole amount of bandwidth required by the task controllers. To this regard, Theorem 3 (claims 1 and 2) provides us with an estimation of the minimum bandwidth $\overline{B}_B^{(i)}$ required for tasks of class B. Likewise, the minimum bandwidth $\overline{B}_A^{(i)}$ required for tasks of class A is the maximum between the one indicated in Theorem 3 (claim 3) and the one indicated in Theorem 2. Clearly, for each task $\tau^{(i)}$, we have $\overline{B}_A^{(i)} \geq \overline{B}_B^{(i)}$.

In order to attain the design goals stated earlier, 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 $B_A^{(i)}$,

- 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 $B_B^{(i)}$,
- 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 V-C.2)

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

$$\sum_{j \in \text{class A}} \overline{B}_A^{(j)} + \sum_{j \in \text{class B}} \overline{B}_B^{(j)} \leq U_{lub} \tag{11}$$

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 III-C 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.

V. Software architecture

In this section we describe a software architecture aimed at providing a concrete implementation of the techniques proposed in the previous sections. As a reference system, we chose GNU/Linux OS (Kernel Version 2.4.27). A considerable advantage of Linux, alongside of the free availability of source code and documentation, is its modular structure that allows one to extend the kernel by simply inserting a module. Our software is comprised of a middleware layer and of a set of modules running in the kernel.

A. 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 run partly at user and partly at kernel level) uses virtual devices, which are commonplace in Operating Systemsof Posix class.
- Backward compatibility: we did not change the API of the Linux Kernel. Therefore, most of preexisting non real-time applications can run without modifications and are taken care of by the Linux scheduler in the background execution time.
- Flexibility: our architecture allows one to easily introduce new control and prediction algorithms, different from those shown in this paper. These algorithms can be run either in user or in kernel space. In the former case, it is possible to use floating point mathematics and third parties math libraries (which is very useful during the prototyping phase). Another possibility offered by user space implementation is to provide specific predictors along with the different applications (whose introduction into the kernel might be untrusted). On the other hand, kernel-space implementation is certainly more efficient and preferable when the algorithms are well-tested and reliable (as is the case of the algorithms proposed in this paper).

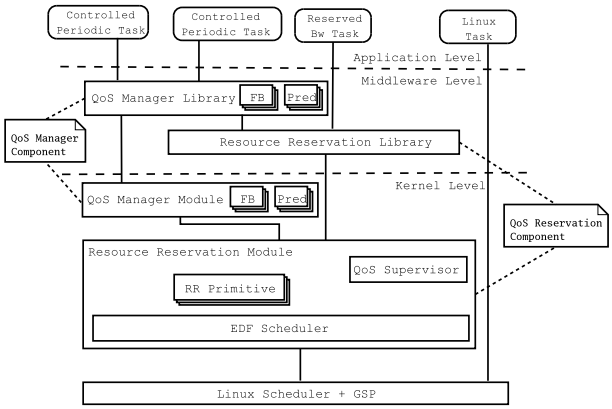

Fig. 4. System Architecture. At the lowermost level we find the linux kernel enriched with the GSP patch. The Resource Reservations algorithm is implemented inside a kernel module, while the management strategies can be implemented partly at the kernel level and partly at the middleware level.

- Efficiency: the overhead introduced by QoS management mechanisms is acceptable. Moreover, the overhead is negligible in case the tasks do not use the QoS management.
- Security: the possibility of changing the bandwidth reserved to the different applications could potentially allow for denial of service attacks; therefore, specific protection mechanisms must be devised. In particular, the system administrator can define “maximum” CPU bandwidths on a per-user basis (in the same way as disk quotas).

The proposed architecture is depicted in Figure 4, and it is composed of the following main components:

- the Generic Scheduler Patch (GSP), a small patch to the kernel (115 added lines, 7 modified files) which allows us to extend the Linux scheduler functionality by intercepting scheduling events and executing external code in a kernel module;
- the QoS Reservation component, composed of a kernel module and 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 use different 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 subcomponents 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 resource:
  - 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 functional-

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

/* Modification of the linux task struct */
struct task_struct {
  /* Linux standard fields */
  ...
  void *scheduling_data;            /* GSP additonal 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;
  ...
}
```

Fig. 5. Example utilisation of the hook mechanism

ities; 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). In the former case, the library communicates with the Resource Reservation module (using the RR library) 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.

B. The Generic Scheduler Patch

A preliminary description of this patch can be found in [25]. 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. In 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 5 clarifies this concept: whenever a scheduling-related event occurs, the appropriate function is invoked through a function pointer (the hook). Any hook is initialised to NULL, and it can be appropriately set by a dynamically loadable scheduling module. The relevant events for which hooks have been

introduced are: task creation, termination, unblock and block. For each of these, we defined the corresponding hook pointer: `fork_hook`, `cleanup_hook`, `unblock_hook` and `block_hook`.

The handlers functions and the necessary data structures (e.g., scheduling queues) are contained in a kernel module. The hook handlers receive the pointer to the `task_struct` of the interested task as a parameter. The GSP patch also allows one 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 associate each task to the server it is currently running into.

C. The QoS Reservation component

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 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.

The Resource Reservation module forces the Linux scheduling decisions by using the `SCHED_RR` policy. When a scheduling module selects the task to be executed according to the Resource Reservations rules, it sets the real-time scheduling priority of the task to the maximum possible value, and then invokes the standard Linux scheduler. On the contrary, when our scheduler needs to prevent a task from being executed, it sets its real-time priority to a value below the minimum possible value and again it invokes the Linux scheduler (in this case the task is blocked). Clearly, linux applications cannot use real-time priorities to avoid interference with this mechanism, but this is not to be seen as a limitation for we provide explicit support to time sensitive applications through the Resource Reservations algorithm.

This approach minimised 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 [36].

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 the application by a library contained in the middleware layer (Resource Reservation library). The communication between the Resource Reservation library and the Resource Reservation module is established through a Linux virtual device.

1) Resource Reservation module: This module implements various resource reservation algorithms: CBS [4], IRIS [31] and GRUB [19].

When the module is inserted the appropriate hooks are set and the data structures are initialised (see Figure 5). Linux tasks that do not need resource reservations are still managed by the Linux scheduler with the SCHED_OTHER policy. Tasks that use the Resource Reservation mechanism are scheduled by our module. The module internally implements an EDF queue and uses kernel timers 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 the CPU fraction used by reserved tasks may starve standard Linux tasks and system services.

2) The QoS Supervisor: The bandwidth requests coming from the task controllers can be accepted, delayed, reshaped or rejected by the QoS Supervisor. Indeed, because the supervisor is endowed with a global vision of the CPU partitioning at any point in time, it is able to enforce the global consistency relation in Equation (1). In addition to that, 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 for each group of users.

To fulfil these complex goals, the supervisor operates in different modes: at the creation of a task and during its execution. When an application wants to create a new server, the request passes through the QoS Supervisor module, which performs the admission control. In particular, in order for our feedback scheme to work properly, the supervisor is required to enforce the condition in Equation (11) that poses a bound to the QoS requirements of tasks in the system. Along with this test, the supervisor also checks that the security rules are not violated (e.g., checking that the minimum guaranteed bandwidth does not exceed the maximum bandwidth granted to the user that runs the task). If the request is accepted, the server is created and the application is guaranteed the requested QoS parameters. Otherwise, an error value is returned to the application. At run-time, as we discussed in Section IV-B, the QoS Supervisor can propagate to the scheduler the bandwidth requests from the task controllers (if they do not violate security rules) when the system is not overloaded. In overload conditions, tasks of class A and B have to receive a minimum guaranteed amount of bandwidth when they require it. Regarding the extra bandwidth (i.e., the difference between U_{lub} and the bandwidth that the guaranteed tasks require), the supervisor uses an allocation heuristic based on *priority levels* and *weights*. First, the available bandwidth is assigned to requests from tasks with higher priority levels. Among tasks in the same level, the bandwidth is shared in proportion to their weights. A privileged user can define priority levels and weights on a per-user and a per-group basis. Moreover, maximum per-level bandwidth 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. To facilitate the system

configuration, the administrator can write a configuration file which is parsed upon system start-up by a privileged program, which translates the file into the appropriate API calls.

3) Resource Reservation library: The Resource Reservation library exports the functionalities 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.

D. The 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 subcomponents, the library redirects all library calls to either user-space code or kernel-space management code (through the proper virtual device).

In Figure 6 we report a sequence diagrams that shows the sequence of function called when a task 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 the qmgr_end_cycle() which is redirected to the appropriate function (for kernel-space implementation, the primitive is translated into a `ioctl()` invocation to 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 only one switch is needed. On the contrary, when the controller is implemented in user space we need two switches (the first one to get the sensor data and the second one to require request the bandwidth change).

For some hardware architectures, the difference in overhead between the two implementation can be negligible, in which case the user space implementation is preferable for its flexibility. On embedded systems, the additional overhead of the user-space implementation might be considerable.

E. Task structure

The typical structure of a task making use of our middleware for the adaptive reservations is shown in Figure 7. For the sake of flexibility, the mechanism required to ensure a periodic activation of the task is left to the appli-

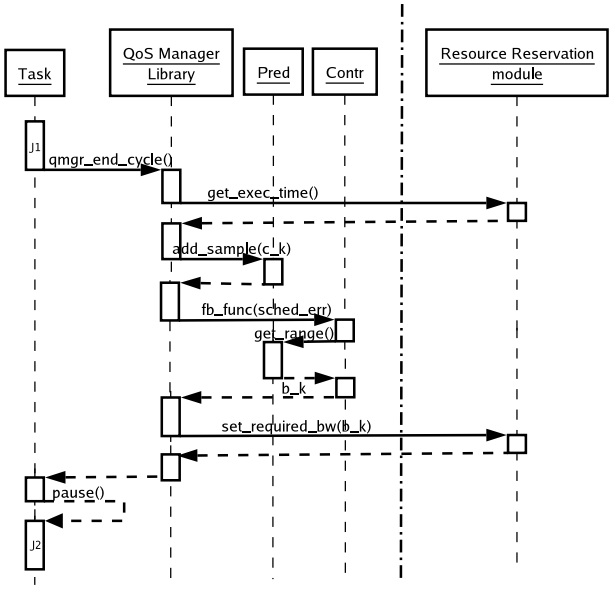

Fig. 6. Interaction among various parts of the architecture when controller and predictor are in user space. The communications with Resource Reservation kernel module uses a virtual device. For the sake of simplicity, we omit the RR library that forward the request to the RR kernel module.

cation programmer. As an aid to programmers, the Qos Manager library also offers the qmgr_start_periodic() utility function. This function receives as a parameter a pointer to the function that implements a job, and ensures that such function will be periodically executed and terminated by the qmgr_end_cycle() invocation.

Of particular interest is the library functions that selects the control and prediction algorithms. The user is allowed to specify his/her own user-level implementation for such algorithms or to choose among one of the library components. For the moment being, the only available control algorithm is the one shown in this paper, which requires the following parameters: the boundaries e, E of the required RCIS, the maximum number of steps h to return into the invariant set from a maximum deviation ΓP and the saturation value for the bandwidth (set to the maximum bandwidth allowed to the user owning the task by default). The provision of these parameters (for which default values are available) is enough to specify the behaviour of the controller in a best effort framework (i.e., for a Class C task). If we want a guaranteed behaviour (Class A or Class B), we also have to provide a minimum guaranteed bandwidth, which is set to 0 if not otherwise specified, and a predictor accurate enough for the specification of the RCIS. These data can be cached in case of repeated play of the same contents (e.g., the streaming of a movie in a video-on-demand system). This is clearly impossible for applications running in real-time (e.g., a video conference) or for applications executed for the first time. For this reason, our system supports the mechanism of *trial executions*, i.e. the ability to run an application with no guarantees or with a fixed bandwidth for a limited period of time, in order to compute the QoS parameters needed by the middleware. These parameters may be retrieved by the application at the end of the trial execution by means of a library call.

As a final remark, if one does not need to use the adaptation mechanism, it is possible to transparently use

```
/* Task structure */

void my_task() {
  /* Initialise the Qos Manager and check if the modules are in the kernel */
  qmgr_init();

  /* set task parameters: RR server period, maximum guaranteed bandwidth, etc. */
  qmgr_set_task_period(TASK_PERIOD);
  qmgr_set_server_period(SERV_PERIOD);
  qmgr_set_max_bandwidth(SERV_MAX_BW);

  /* set the predictor and controller component and customise their 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() != 0)
  {
    /* 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 {
    /* manage supervisor error condition */
    ...
  }
  /* release task data structures and stop feedback loop */
  qmgr_end();
}
```

Fig. 7. Example task code

the Resource Reservations mechanism. In this way, legacy applications can be executed with fixed bandwidth without modification of the structure. The idea is to create a task in a server and then issue an `exec` call on the legacy code.

VI. 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 [17] 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 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.

As we said earlier, our software is developed on the top of the Linux Kernel Version 2.4.27. Together with the GSP patch, we applied to the kernel two additional patches: the High Resolution Timer (hrtimers-2.4.20-3.0; [21]) and Linux Trace Toolkit (TraceToolkit-0.9.5; [29]).

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

TABLE I

HARDWARE-SOFTWARE PLATFORM.

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 (10ms for kernel 2.4). Using the High Resolution Timers patch, it is possible to set timers with a granularity in the order of ten microseconds.

The LTT patch is a powerful kernel-tracing system developed for Linux kernel. Namely, it allows one to exactly log the occurrence of any kernel-related event (system calls, interrupts, scheduling events etc.). 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.

A. 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 discusses 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 [19]). 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

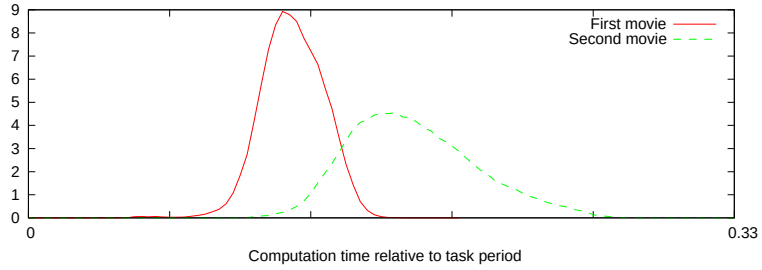

Fig. 8. Experimental probability mass function (PMF) of the input traces

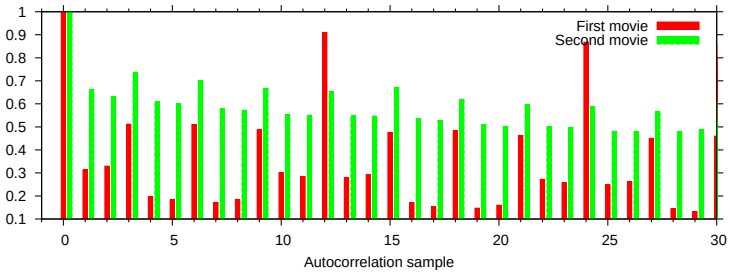

Fig. 9. Autocorrelation for the two used input traces

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 8). For both we used an adaptive reservation with period $P = 1ms$, which is $1/40^{th}$ of the task period.

The difference in the stochastic properties of the processes is due to the coding scheme. Indeed, an MPEG-2 file consists of frames belonging to three classes: I, P, B. It is very frequent in DVD to have a periodic structure (e.g., *IBBPBBPBBPBBI...*). This situation is referred to as *closed group of pictures (GOP)*. This is exactly the case for the first video stream in which frames of type I are repeated with period 12. This is visible in the autocorrelation structure (see Figure 9), which displays important peaks repeated with period 12. There is also a sub-period (due to the repetition of P frames), which determines smaller peaks every three samples. The second MPEG file has a more complex structure. In this case, the coding scheme is piecewise periodic (i.e., frames of type *I* are more frequent in certain scenes). In other words, we do not have a closed GOP. In particular, in certain segments we identified a periodicity of 15. The autocorrelation function (which is averaged throughout the whole trace) does only display a significant peak for the third sample (see Figure 9)[2].

For the first video stream, the average and the maximum computation bandwidth required through the whole

[2]If we had a prior knowledge of the different periods used in a piecewise periodic scheme, it could be possible to use adaptive predictors able to recognise the periodicity changes and adjust the algorithm accordingly. This type of sophisticated predictors is out of the scope of this paper.

trace were respectively $B_{av1} = 18.33\%$ and $B_{max1} = 62.31\%$. In this number we also accounted 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 unmodeled 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 execution of 1000 samples showed that a maximum bandwidth of 25% was sufficient to meet the requirements.

In Figure 10, we compare the experimental PMF attained throughout the whole execution. For the sake of completeness we also show the PMF obtained with a fixed bandwidth for two values: 20% and 25%. In Table II we report the quantitative evaluation of the performance (using the metric introduced above). With a static bandwidth allocation "centring" the distribution inside the target set is not an easy task. Indeed, a bandwidth value too close to average results into local instabilities throughout the execution (leading to enormous average errors). Conversely, a high bandwidth value ensures stability of the system, but the average scheduling error is negative and high in modulus, highlighting a wasteful usage of the CPU resource.

Looking at the feedback schemes. The controller endowed with the MMA predictor is able to compensate the different computation times for frames of different types. Therefore, the second peak of the PMF disappears and the probability of staying in the target set is 76 %. The improvement attained with the OL predictors is evident in the picture and in the quantitative figures. The price to pay is clearly the additional complexity. In most cases, for all predictors, a deviation from the target set is recovered in one step. Indeed, for instance, the average number of steps required to restore the equilibrium with the MMA[3,12] predictor was 1.034.

For the second video stream, the average and the maximum computation bandwidth required through the whole trace were respectively 12.65% and 20.33%. In this case, the specification for robust controlled invariant set was $e = -0.2T$ and $E = 0.05T$. In case of deviation from the equilibrium, we required recovery in 3 steps. As far as predictors are concerned, for this experiment we used a MA[3],MMA[3,3] and OL[15,180]. Also for this case, a maximum bandwidth equal to 25% was estimated to be sufficient for attaining the specification in a trial execution of 1000 samples.

The experimental PMF are reported in Figure 11 and the quantitative data in Table II. Due to the lack of a strong correlation structure in this trace, the improvement using sophisticated predictors is not as large as for the first video stream. The average number of steps required to restore the equilibrium with the MA[3] predictor was 1.142.

B. Overhead evaluation

The overhead introduced by our middleware is the combination of two different effects:

- the time required to execute the code of the middleware functions (resource reservation and control mechanisms).
- the number of additional context switches introduced by the scheduler.

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%

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.

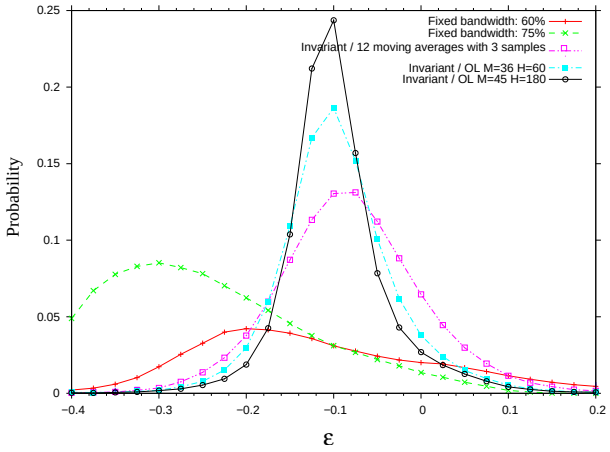

Fig. 10. Comparison among the PMF achieved by various controllers and predictors on the first video stream

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.

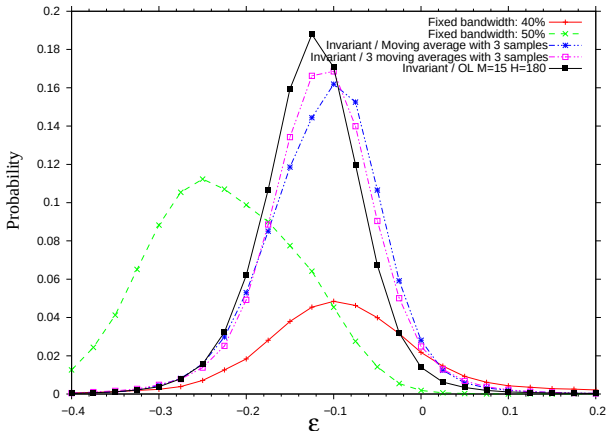

Fig. 11. Comparison among the PMF achieved by various controllers and predictors on the second video stream

	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

TABLE III

MEASURED EXECUTION TIME IN CPU CLOCK CYCLES, OF THE SCHEDULING HOOKS. FOR COMPARISON, WE ALSO REPORT THE EXECUTION TIME OF THE STANDARD LINUX FUNCTIONS DO_FORK, DO_EXIT AND SCHEDULE, WHICH ARE EXECUTED *in addition* TO OUR CODE.

We evaluated the overhead effects on the MPEG decoding application (using as input the first video stream cited above).

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 task. 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 two fixed values for the bandwidth respectively $0.9B_{av1}$ (i.e., 10% lower than the average required bandwidth) and $1.1B_{av1}$ (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. The semantics of the hook handlers is reported in Section V. 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 5), 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_timout()` function is the handler of timer-related event: 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 grow 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()`.

The second experiment that we performed was aimed at evaluating the number of context switches and the total overhead introduced by our mechanisms. In particular 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 referring to its true computation time (i.e., task 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 12. Namely, for each workload situation, we varied the server period (on the horizontal axis). In the first two rows, we report the results for fixed bandwidth and in the third row result for 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 grow rate of both quantities is approximatively linear with the number of dummy tasks.

The changes with respect to the reservation period follow approximatively 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

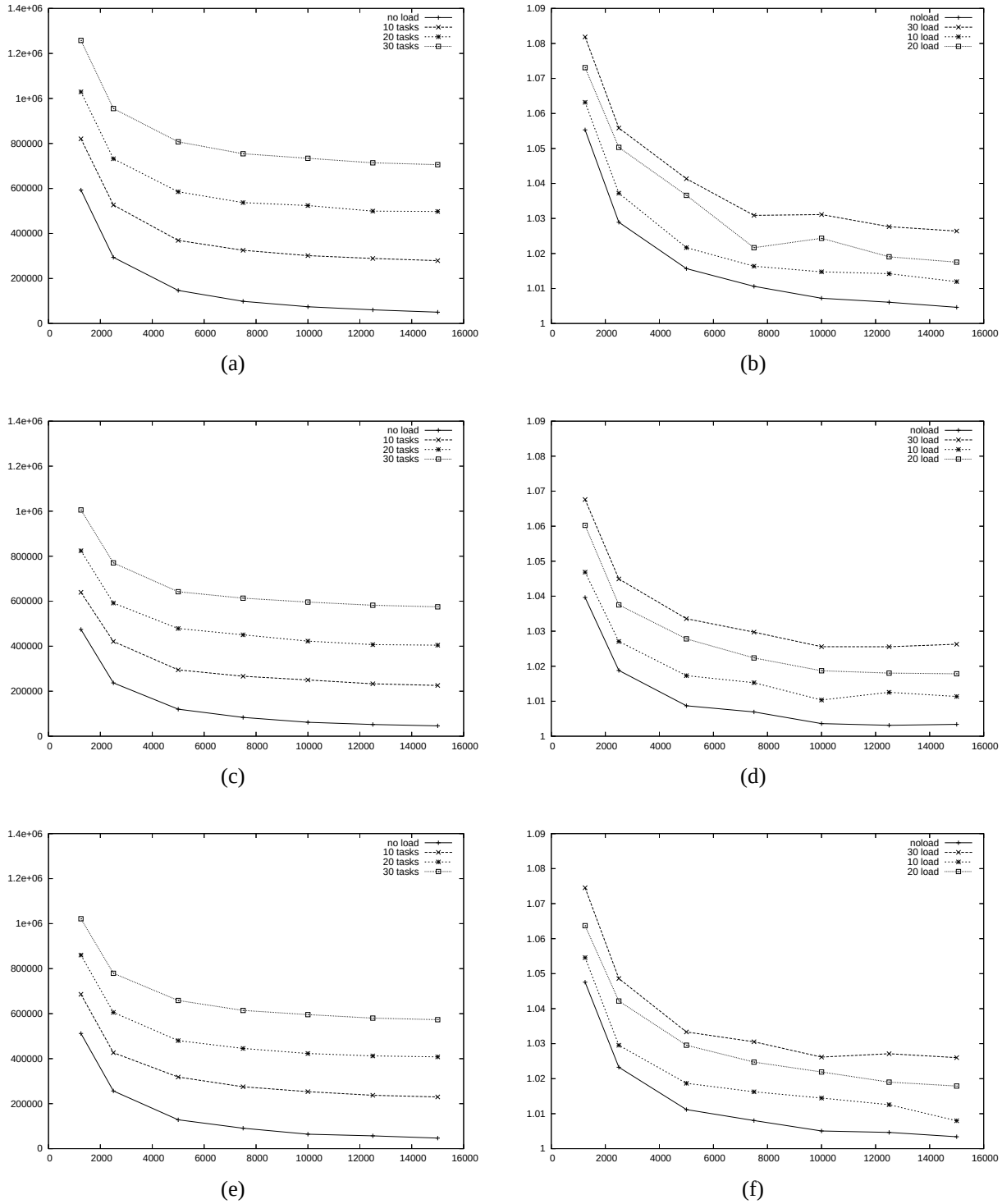

Fig. 12. Number of context switches (left column) and relative overhead measure on the first video stream (right column). For all plots, the horizontal axis represent the server period P. (a), (b) are related to a fixed bandwidth equal to $0.9B_{av1}$. (c), (d) are related to a fixed bandwidth equal to $1.1B_{av1}$. (d), (e) are related to the dynamic allocation using an MA[3] predictor.

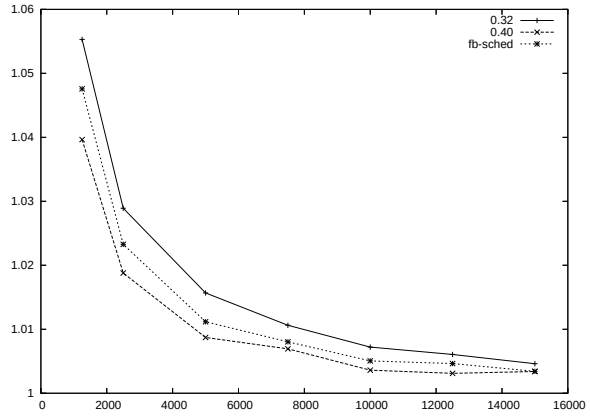

Fig. 13. Comparison of the total relative overhead for fixed and dynamic bandwidth (no dummy tasks)

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 we explained above, one would expect a lower overhead using a greater bandwidth. And this fact is shown in Figure VI-B, where the measurements have been taken in absence of dummy tasks. As it is possible to see, the values 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 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).

VII. 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 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, frame buffer). 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).

APPENDIX

Proof: Theorem 2

Let $\hat{\varepsilon}_k$ be defined as $\frac{\varepsilon_k}{P}$, and let $\hat{\mathcal{I}}$ be the interval $[-\hat{e}, \hat{E}]$. $\mathcal{I}$ is a RCIS iff there exists a control law ensuring $\hat{\varepsilon}_{k+1} \in \hat{\mathcal{I}}$ whenever $\hat{\varepsilon}_k \in [-\hat{e}, \hat{E}]$. System evolution of Equation 5 may then be restated as:

$$\hat{\varepsilon}_{k+1} = S(\hat{\varepsilon}_k) + \left\lceil \frac{c_k}{q_k} \right\rceil - L. \tag{12}$$

Then, $\hat{\varepsilon}_{k+1} \in \hat{\mathcal{I}}$ iff $L - \hat{e} - S(\hat{\varepsilon}_k) \leq \left\lceil \frac{c_k}{q_k} \right\rceil \leq L + \hat{E} - S(\hat{\varepsilon}_k)$. Considering that $\forall a, b \in \mathbb{Z}$, $\forall x \in \mathbb{R}$ $a \leq \lceil x \rceil \leq b \iff a - 1 < x \wedge x \leq b$, it is possible to write: $\hat{\varepsilon}_{k+1} \in \hat{\mathcal{I}} \iff q_k \in \left[\frac{c_k}{L+\hat{E}-S(\hat{\varepsilon}_k)}, \frac{c_k}{L-1-\hat{e}-S(\hat{\varepsilon}_k)} \right[\triangleq \mathcal{Q}(c_k, \hat{\varepsilon}_k)$. The latter condition is satisfied $\forall c_k \in [h_k, H_k]$ if and only if $q_k \in \bigcap_{c_k \in [h_k, H_k]} \mathcal{Q}(c_k, \hat{\varepsilon}_k) = \left[\frac{H_k}{L+\hat{E}-S(\hat{\varepsilon}_k)}, \frac{h_k}{L-1-\hat{e}-S(\hat{\varepsilon}_k)} \right[$. Considering the saturation constraint $q_k \leq \overline{Q}$, we get that the set of control values ensuring robust control invariance of $\mathcal{I}$ is given by:

$$Q_k \in \overline{\mathcal{Q}}(\varepsilon_k) \triangleq \left[\frac{H_k}{L + \hat{E} - S(\hat{\varepsilon}_k)}, \min\left\{ \frac{h_k}{L - 1 - \hat{e} - S(\hat{\varepsilon}_k)}, \overline{Q} \right\} \right[,$$

which proves claim 2. To have a necessary and sufficient condition for the existence of the RCIS, we need to check for non-emptyness of the set $\overline{\mathcal{Q}}(\varepsilon_k)$. $\overline{\mathcal{Q}}(\varepsilon_k) \neq \emptyset \iff \frac{H_k}{L+\hat{E}-S(\hat{\varepsilon}_k)} < \min\left\{ \frac{h_k}{L-1-\hat{e}-S(\hat{\varepsilon}_k)}, \overline{Q} \right\} \iff \hat{e} + \alpha_k \hat{E} > (1-\alpha_k)(L - S(\hat{\varepsilon}_k)) - 1 \wedge \overline{Q} > \frac{H_k}{L+\hat{E}-S(\hat{\varepsilon}_k)}$. Imposing this condition for all $\varepsilon_k \in [-\hat{e}, \hat{E}]$, we get: $\overline{\mathcal{Q}}(\varepsilon_k) \neq \emptyset \iff \hat{e} + \alpha_k \hat{E} > (1-\alpha_k)L - 1 \wedge \overline{Q} > \frac{H_k}{L}$. The proof of Claim 1 is completed observing that the above condition has hold starting from any step k_0. ■

Proof: Theorem 3

As we did before, define $\hat{\varepsilon}_k = \frac{\varepsilon_k}{P}$.

Proof of Claim 3. For our purposes we can consider only the case when the scheduling error remains positive all along the interval $[k, k+h-1]$. From Equation 12 it follows that the minimum number of steps to go from the fartherst point of $\mathcal{J}$ to $\mathcal{I}$ is: $h \geq \left\lceil \frac{\Gamma - \hat{E}}{L-1} \right\rceil$. Let's start proving sufficiency. As a particular choice, the controller can always choose $\overline{Q}$ until the state reaches the target set. After h steps, it is possible to write:

$$\hat{\varepsilon}_{k+h} \leq \hat{\varepsilon}_k + \sum_{j=k}^{k+h-1} \left\lceil H_j / \overline{Q} \right\rceil - Lh \leq \hat{\varepsilon}_k + \frac{\tilde{H}_h}{\overline{Q}} h - Lh \leq \Gamma + \frac{\tilde{H}_h}{\overline{Q}} h - Lh.$$

Hence, the target set is always reached in at most h steps if: $\Gamma + \frac{\tilde{H}_h}{\overline{Q}} h - Lh \leq \hat{E}$ and the claim easily follows. Necessity can be proven by contradiction. Assume that choosing $\overline{Q} < \frac{h\tilde{H}_h}{hL - \Gamma + \hat{E}}$ it is possible to reach $\mathcal{I}$ in h steps. Then $\overline{Q}(hL - \Gamma + \hat{E}) - h\hat{H}_h = -\delta'$, with $\delta' > 0$. By definition of $\sup$ for any $\delta > 0$ there exists $\overline{k}$ such

that $\forall k, \geq \overline{k}\ |1/h \sum_{j=\overline{k}}^{\overline{k}+h-1} \left\lceil \frac{H_k}{\overline{Q}} \right\rceil \overline{Q} - \tilde{H}_h| < \delta$. Choose $\delta < \delta'/h$. Consider a trajectory starting from $\varepsilon_{\overline{k}} = \Gamma$ and assume that $c_j = H_j$ for $j = \overline{k}, \ldots, \overline{k}+h-1$. The maximum allowed bandwidth for the controller is $\overline{Q}$. Therefore,

$$\varepsilon_{\overline{k}+h} \geq \varepsilon_{\overline{k}} + \sum_{j=\overline{k}}^{\overline{k}+h-1} \lceil H_j/\overline{Q} \rceil - hL > \Gamma + \frac{h(\hat{H}_h - \delta)}{\overline{Q}} - hL = \frac{\delta' - \delta h}{\overline{Q}} + \hat{E} > \hat{E}, .$$

which contradicts the hypotheses.

Proof of Claim 1. Reasoning as for Claim 3, assume that the controller always uses $\overline{Q}$. Proceeding as above we can write:

$$\hat{\varepsilon}_{k+h} \leq \hat{\varepsilon}_k + \sum_{j=k}^{k+h-1} \lceil H_j/\overline{Q} \rceil - Lh \leq \Gamma + \sum_{j=0}^{k+h-1} \lceil H_j/\overline{Q} \rceil - Lh.$$

According to the definition of limit, $\forall \delta > 0\ \exists \tilde{h}$ s.t. $\forall h > \tilde{h}$, $|\frac{1}{k+h} \sum_{j=0}^{k+h-1} \left\lceil \frac{H_j}{\overline{Q}} \right\rceil \overline{Q} - \tilde{H}| < \delta$. Therefore for all $h > \tilde{h}$ it holds:

$$\hat{\varepsilon}_{k+h} \leq \Gamma + (\tilde{H} + \delta)\frac{k}{\overline{Q}}(\tilde{H} + \delta - L\overline{Q})\frac{h}{\overline{Q}}.$$

If we choose $0 < \delta < \overline{Q}L - \tilde{H}$, it is second term is decreasing with h. Hence, it is possible to find h big enough as to make $\varepsilon_{k+h} < \hat{E}$.

Proof of Claim 2. Necessity of condition $\overline{Q} \geq \frac{\tilde{H}}{L}$ can be proved by contradiction, following the same line of reasoning as in the proof of Claim 3. Assume $L\overline{Q} - \tilde{H} = -\delta', \delta' > 0$. For global reachability, it must be possible to eventually reduce any sequence into $\mathcal{I}$. In particular this is applicable to the sequence starting for $\varepsilon_k = \Gamma$ in $k = 0$.

Proceeding as in the proof of Claim 3, we can conclude

$$\varepsilon_h \geq \Gamma + \sum_{j=0}^{h-1} \left\lceil \frac{H_j}{\overline{Q}} \right\rceil - Lh.$$

From the definition of limit, $\forall \alpha > 0$, there exists $\overline{h}$ s.t. $\forall h > \overline{h}$ it holds $\tilde{H} - \alpha > \frac{1}{h} \sum_{j=0}^{h-1} \left\lceil \frac{H_j}{\overline{Q}} \right\rceil \overline{Q}$. Hence,

$$\varepsilon_h > \Gamma + \frac{h}{\overline{Q}}(\tilde{H} - \alpha - L\overline{Q}) = \Gamma + \frac{h}{\overline{Q}}(\alpha' - \alpha).$$

If we choose $\alpha < \alpha'$ the sequence diverges, which contradicts our hypotheses.

■

Proof: Fact 1 Consider the system evolution as stated in Equation 12. $\tilde{c} \in \mathcal{Z}$ if and only if $\hat{\varepsilon}_{k+1} = S(\hat{\varepsilon}_k) + \left\lceil \frac{\tilde{c}}{q_k} \right\rceil - L = 0 \iff q_k \in \left[\frac{\tilde{c}}{L - S(\hat{\varepsilon}_k)}, \frac{\tilde{c}}{L - S(\hat{\varepsilon}_k) - 1} \right[$. The controller must be able to choose a q_k which respects the above equation, as well as Equation 8. A non-empty intersection between the two sets results in:

$$H_k \frac{L - S(\hat{\varepsilon}_k) - 1}{L - S(\hat{\varepsilon}_k) + \hat{E}} < \tilde{c} \leq h_k \frac{L - S(\hat{\varepsilon}_k)}{L - 1 - S(\hat{\varepsilon}_k) - \hat{e}} \wedge \tilde{c} \leq \overline{Q}(L - S(\hat{\varepsilon}_k))$$

Intersecting conditions for $\hat{\varepsilon}_k \in [-\hat{e}, \hat{E}]$, the set of 1-step zero controllable $\tilde{c}$ values of Equation 9 is obtained.

■

REFERENCES

[1] 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.

[2] 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.

[3] Luca Abeni. Server mechanisms for multimedia applications. Technical Report RETIS TR98-01, Scuola Superiore S. Anna, 1998.

[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] 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.

[9] 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.

[10] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Boston, 1997.

[11] G. Tao C. Lu, J. Stankovic 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.

[12] Anton Cervin, Johan Eker, Bo Bernhardsson, and Karl-Erik Årzén. Feedback-feedforward scheduling of control tasks. *Real-Time Systems*, 2002. To appear.

[13] 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.

[14] J. C. R. Bennett e H. Zhang. Hierarchical packet fair queueing algorithms. In *P Proceedings of ACM SIGMETRICS '96*, 1996.

[15] J. C. R. Bennett e H.Zhang. wf^2q: Worst-case fair weighted fair queueing. In *Proceedings of IEEE INFOCOM '96*, 1996.

[16] 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.

[17] Ffmpeg project, `http://ffmpeg.sourceforge.net`. Web site.

[18] 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.

[19] 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.

[20] Ashvin Goel, Jonathan Walpole, and Molly Shor. Real-rate scheduling. In *Proceedings of Real-time and Embedded Technology and Applications Symposium*, page 434, 2004.

[21] High resolution timers, `http://high-res-timers.sourceforge.net`. Web site.

[22] 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.

[23] Kevin Jeffay, F.D. SMith, A. Moorthy, and J.H. Anderson. Proportional share scheduling of operating system services for real-time applications. In *IEEE Real Time System Symposium*, Madrid, Spain, December 1998.

[24] 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.

[25] Abeni L. and Lipari G. Implementing resource reservations in linux. In *Real-Time Linux Workshop*, Boston (MA), USA, December 2002.

[26] 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.

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

[28] Jane Liu. *Real-time sytems*. Prentice Hall, 2000.

[29] Linux trace toolkit, `http://www.opersys.com/LTT`. Web site.

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] A. Parekh and R.G. Gallager. A generalized processor sharing approach to flow control - the single node case. In *Proceedings of INFOCOM '92*, 1992.

[36] Preemption patch, `http://kpreempt.sourceforge.net`. Web site.

[37] Ragunathan Rajkumar, Chen Lee, John P. Lehoczky, and Daniel P. Siewiorek. Practical solutions for QoS-based resource allocation. In *RTSS*, pages 296–306, 1998.

[38] 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.

[39] Dickson Reed and Robin Fairbairns (eds.). Nemesis, the kernel – overview, May 1997.

[40] 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.

[41] Ali H. Saye. *Fundamentals of Adaptive Filtering*. Wiley-IEEE Press, June 2003.

[42] 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.

[43] 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.

[44] 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.

[45] Paolo Valente. Exact gps simulation with logarithmic complexity, and its application to an optimally fair scheduler. In *Proceedings of ACM SIGCOMM'04*, 2004.

[46] 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.

[47] 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.