Hierarchical QoS Management for Time Sensitive Applications

Luca Abeni
RETIS Lab
Scuola Superiore S. Anna, Pisa
luca@sssup.it

Giorgio Buttazzo
University of Pavia (Italy)
INFM - Pavia research unit
giorgio@sssup.it

Abstract

The use of real-time techniques in new application fields, such as multimedia computing, has extended classical algorithms to more dynamic environments, introducing the problem of controlling and adapting the quality of service provided by an application.

In this paper, we investigate the possibility of integrating application-dependent adaptation strategies with reservation techniques for handling multimedia real-time applications. We show how a global adaptive reservation mechanism can be combined with a local application-level adaptation, obtaining a hierarchical management scheme.

The need for the two forms of adaptation (application dependent and global) and the effectiveness of the proposed hierarchical scheme are shown by a set of experiments on multimedia applications. All the experiments have been performed by running the real applications on a real-time kernel.

1. Introduction

In the last years, a lot of work has been done to show that Quality of Service (QoS) adaptation can be useful to support time sensitive applications (like multimedia ones) in a general purpose workstation.

The basic idea behind QoS adaptation (borrowed from the network community) is to avoid overload situations by scaling down the applications' resource requirements. This can be done using two different approaches:

- in a first approach, the adaptation strategy is embedded in the application, so that each task is locally responsible for changing its computational demand based on the experienced service;
- in a second approach, the QoS adaptation is usually performed by a global QoS manager, which "suggests" the application tasks to change their computational demand based on the current system workload.

Both the two approaches can be implemented without any specific support from the system. However, if the system is not able to bound the resource requirements of each application, both solutions might not be very effective. In fact, the overall system performance can hardly be optimized using local information only, and a misbehaved task requiring too much processor time could significantly degrade the other activities. On the other hand, a global approach alone can also fail in the presence of applications that do not properly respond to the QoS manager "suggestions". For example, a non adaptive task requiring too much resources could force the other tasks to adapt their QoS to the minimum value.

The problems mentioned above can be solved by bounding the resource requirements of each application through a scheduling mechanism providing temporal protection, such as resource reservations or proportional share scheduling. However, integrating such a reservation mechanism with the application-level QoS adaptation is still an open issue.

In this paper, we propose a hybrid method for integrating a reservation mechanism with an application level QoS adaptation strategy. The method controls the CPU bandwidth reserved to a task, but allows each task to change its QoS requirements if the amount of reserved resources is not sufficient to accomplish the goal within a desired deadline. Using such an integrated approach, the QoS adaptation is performed in an application-specific fashion: each application can react to overloads in a different way and use different techniques to scale down its resource requirements. On the other hand, if an application does not adequately react to a resource lack, the scheduler will slow it down in order not to influence the other applications.

Two possible strategies can be used to allocate resources to an application: the first one is based on a fixed reservation, where resources are statically assigned to tasks at system initialization; the second is based on a dynamic adaptation of the reserved resources. Both techniques can be combined with the proposed QoS adaptation methodology. In the second case, the scheduler (or a QoS manager) automatically adapts the amount of reserved resources to the application requests, independently from application semantics, whereas the application scales its QoS requirements (and consequently, its requests) in response to the assigned amount of resources.

1.1. Related work

A lot of research has been done to extend classical real-time results [12] to time sensitive applications with timing characteristics that are different from the ones found in typical control systems, for which such techniques were originally developed. To apply such techniques in more dynamic environments, where arrival times and computation times may have significant variations, a resource reservation (RR) approach [15, 11, 6, 1] has been proposed, based on the concept of *temporal protection*.

Temporal protection (also known as temporal *isolation*) requires that the temporal behavior of a task is not influenced by the other tasks in the system and each task runs as if it is executing alone on a slower processor. Using this approach, each task can be guaranteed independently from the others: it is possible to check a-priori if a task will respect its temporal requirements, independently from the amount of resources required by all the other tasks.

A different approach to achieve similar results is represented by Proportional Share (PS) scheduling [8, 18]. In [3] it is shown that the RR and the PS approaches provide different interfaces for similar services (although RR enforces more control on the provided QoS). Some work, like [19], propose a trade-off between RR and PS techniques.

In the last years, the need for some form of adaptation in the QoS provided by an application has emerged. Some researcher [7] claims that such an adaptation can be achieved without any specific support from the operating system, while others tend to provide an explicit support in the scheduler, through an active entity (the QoS manager) [14, 4, 6], developed on top of the kernel.

In [6], the scheduler implements a form a resource reservation based on the Earliest Deadline First (EDF) algorithm [12], and can detect variations in the application requirements in order to adapt the reservations to the new system workload. However, this adaptation is performed in a global context and does not consider the application behavior.

In [4], each application QoS can be scaled by a global QoS manager in order to better respond to the user needs. The adaptation is based on specific *modes of operation* provided by each application, but it is still performed on a global base.

In [16], a formal definition of QoS is presented, and an algorithm to allocate system's resources for maximizing the total QoS is described. This work is extended in [10], considering multiple QoS dimensions: each application is characterized by multiple QoS dimensions (such as frame rate, format, depth, compression type, and so on) and requires multiple resources (RAM, CPU time, disk, and so on) to reach each specified QoS level. The user is allowed to specify the utility obtained by the application on each QoS dimension through a proper *utility function*, while the QoS requirements can be mapped to resource usage using a set of profiles. In this way, using some dynamic programming and mixed integer programming techniques, it is possible to compute an optimal resource assignment which maximizes the total utility. The problem with this approach is that it cannot be applied when the QoS mapping profiles (used to map the QoS specification to resource usage) are unknown. In this case, we believe that the only way to control the QoS experienced by each task is to use some form of feedback (implicitly reconstructing the QoS mapping profiles on-line).

In [5], the authors present a feedback-based QoS manager, DQM, which does not require any support from the operating system. DQM is a middleware solution aimed at supporting soft real-time applications in a conventional OS (Linux). The DQM middleware can change some applications' execution level based on resource usage and benefit provided by the application and based on the system load estimated by the middleware itself. A similar approach, based on a resource allocator that monitors the resource usage and coordinates the adaptation, is presented in [17]. This solution addresses the problem of integrating QoS adaptation with real-time techniques, but depends on the a-priori knowledge of the resources required by each application in each operating mode.

In [14], a solution based on resource reservations and on a global QoS manager is presented. In particular, the QoS manager detects reservation overruns and adapts the tasks' periods and computation times in order to reduce the overruns. However, it is not clear how the QoS manager can control the behavior of each single application.

The use of feedback schemes in real-time scheduling is emerging in recent research: for example, in [13], the number of missed deadlines is used as a feedback to control the system workload by a proper admission control policy. The proposed method, however, does not provide isolation among tasks, thus computational demands can hardly be controlled individually.

A feedback mechanism for adapting the scheduler parameters is presented in [2]. In this paper, a reservation approach is used, and the feedback is used to adapt the fraction of CPU bandwidth reserved to each task. The bandwidth adaptation mechanism is global and weights are used to assign a different importance to each task, independently from its demanded resources. However, the QoS of each task, expressed by its period (or its desired activation frequency) is not changed by a QoS manager, but is adapted by the task itself.

The main contribution of this work is to show how a static or adaptive resource reservation scheme can be combined with application level QoS management, in order to obtain a QoS adaptation mechanism which protects the adaptive applications from the misbehaved ones (as discussed in the introduction, most of the previously proposed QoS adaptation schemes fail in achieving this last goal).

2. QoS adaptation model

For some application, it is important to have the possibility to control its QoS locally, without any global QoS manager, since each activity may react to system overloads in different ways.

On the other hand, although some authors present a form of QoS control that does not require any support from the OS, we believe that such a support is required to avoid instability in the adaptation, and to avoid that misbehaved applications affect well-behaving ones.

The goal of this paper is to integrate resource reservation with application-level QoS management. Moreover, we introduce a two-level (hierarchical) adaptation architecture: a local (application-level) QoS adaptation and a global (system-level) adaptation of the amount of resources reserved to each task. One of our design goals is to decouple as much as possible these two adaptation loops.

In the next section we briefly recall the resource reservation approach and show when QoS adaptation needs to be performed.

2.1. Definitions

A task τ_i is a stream of requests, or jobs, $J_{i,j}$, ($j = 1, \ldots, n$), each characterized by an arrival time $r_{i,j}$, an execution time $c_{i,j}$, a finishing time $f_{i,j}$, and a deadline $d_{i,j}$. Tasks can be *hard* or *soft*: a task τ_i is said to be hard if all its instances have to complete within their deadline ($\forall j \quad f_{i,j} \leq d_{i,j}$), otherwise a critical failure may occur in the system. A task is said to be soft if a deadline miss in one or more instances is tolerated by the system.

The sequence of the execution times $c_{i,j}$ of task τ_i can be seen as a sequence of values distributed according to a Probability Distribution Function (PDF) $V(c) = P\{c_{i,j} = c\}$, or as a stochastic process. Since $V(c)$ does not depend on j, the stochastic process is stationary and time-invariant, so it is ergodic. Hence, the execution times expectation $E[c] = \sum cV(c)$ is equal to the mean execution time, computed as $\overline{c} = \lim_{k\to\infty} \frac{\sum_{j=1}^{k} c_{i,j}}{k}$.

A task is said to be *periodic* if $r_{i,j} = (j-1)T_i$, where T_i is the task period. Although QoS adaptation can be applied to every kind of task, for the sake of simplicity in this work we will consider only periodic tasks. We also consider tasks characterized by soft deadlines, and with a relative deadline equal to the task period: $d_{i,j} = r_{i,j} + T_i$, hence $d_{i,j} = jT_i$.

We define the bandwidth *required* by a task τ_i as

$$B_i^{req} = \lim_{k\to\infty} B_i^{req}(0, k) \tag{1}$$

with

$$B_i^{req}(0, k) = \frac{\sum_{j=1}^{k} c_{i,j}}{kT_i}.$$

In this work, tasks are handled by dedicated servers and scheduled by EDF. When $J_{i,j}$ is released at time $r_{i,j}$, the job is assigned a *scheduling deadline* and it is inserted in a queue ordered by scheduling deadlines. Then, the first job of the queue (the one with the shortest scheduling deadline) is scheduled to execute. As we will see in the next section, the server can change the scheduling deadline during the job execution: in this case the EDF queue has to be reordered.

2.2. Resource reservation

Resource reservation techniques have been developed to isolate the temporal behavior of an application from the demand of the others. Using this approach a task is reserved Q_i time units each T_i: in this way, if the guarantee condition

$$\sum_i \frac{Q_i}{T_i} \leq U_{lub} \tag{2}$$

is verified (where U_{lub} is the utilization bound of the scheduling algorithm), each task is guaranteed to execute at least for its reserved amount of time, independently from all the other tasks. If a task requires too much execution time, it will not steal the time reserved to the other tasks in the system.

Resource reservation can be enforced either in static priorities schedulers (such as Rate Monotonic), as in Real-Time Mach, or in dynamic schedulers (such as EDF). In this work we will assume that each task is served by a dedicated Constant Bandwidth Server (CBS) [1], a particular reservation technique based on EDF, which assigns each task a suitable *scheduling deadline* so that the requested bandwidth is never exceeded.

A CBS is described by two parameters: Q_s, *the server maximum budget*, and T_s, the *server period*. The *server bandwidth* $B_i = \frac{Q_s}{T_s}$ is the fraction of the CPU bandwidth reserved to the task served by the CBS. When a job of the served task arrives, the server checks whether the last assigned *scheduling deadline* d_s can be used: in this case, that deadline is assigned to the job, otherwise a new scheduling deadline $d_s = r_{i,j} + T_s$ is generated and assigned to the job. Each time a job executes for Q_s time units, the deadline is postponed by T_s time units. A more complete description of the CBS, with properties and performance results, can be found in [1].

As shown in [2], the CBS provides a simple parameter that can be used as a feedback to check whether the server task is requiring more than the reserved bandwidth. This value is the *CBS scheduling error* $\epsilon_i = d_s - (r_{i,j} + T_i)$, defined as the difference between the last scheduling deadline assigned by the server to the task and the soft deadline. Since the scheduling deadline d_s is postponed when the task requires more than the reserved bandwidth, the scheduling error ϵ_i is greater than 0 if and only if the task is requiring more than the reserved bandwidth ($\epsilon_i > 0 \iff B_i^{req} > B_i$).

2.3. Need for QoS adaptation

As stated above, the CBS ensures that a served task will never demand more than the reserved bandwidth, and if $\sum \frac{Q_i}{T_i} \leq 1$ each task will execute for the reserved bandwidth. If a task requests more than its reserved bandwidth, it will slow down in order not to jeopardize the others.

In [3] it is shown that, if $Q_s > E[c_i]$ the task schedule is predictable, and it is possible to guarantee that each job $J_{i,j}$ will finish within a probabilistic deadline $r_{i,j} + \delta$ with a probability $X(\delta)$. On the contrary, if $Q_s \leq E[c_i]$, the scheduling deadlines assigned to τ_i will diverge to infinity (in order to preserve the bandwidths allocated to the other tasks') and nothing can be guaranteed about $f_{i,j}$.

We now show that the $Q_s > E[c_i]$ constraint can be expressed in terms of reserved and requested bandwidth:

$$Q_s > E[c_i] \quad \Rightarrow \quad Q_s > \overline{c_i} \quad \Rightarrow \quad \frac{Q_s}{T_s} > \frac{\overline{c_i}}{T_s}.$$

If we select $T_s = T_i$, we obtain

$$B_i > \frac{\lim_{k\to\infty} \sum_{j=1}^{k} \frac{c_{i,j}}{k}}{T_i} \quad \Rightarrow$$

$$\Rightarrow \quad B_i > \lim_{k\to\infty} \frac{\sum_{j=1}^{k} c_{i,j}}{\sum_{j=1}^{k} kT_i} \quad \Rightarrow \quad B_i > B_i^{req}.$$

Hence, if a task "requests too much bandwidth" (i.e., if the requested bandwidth is greater than the reserved bandwidth) it has to decrease its computational demand in order to achieve predictability in its schedule. We define a task requiring too much bandwidth as an overloaded task. Task τ_i is said to be overloaded if

$$B_i^{req} \geq B_i. \tag{3}$$

When a task is overloaded, the application can scale down its QoS (and consequently its resource requests), in order to make $B_i^{req} < B_i$, thus removing the overload condition.

3. Performing the QoS adaptation

When a task requires more than the reserved bandwidth, it has to scale down its resource usage. Numerous solutions have been proposed in literature and are well known in the multimedia community; as an example, in this paper we consider two of them: enlarging the task's period, and skipping some tasks' instances.

3.1. Scaling tasks' periods

The first solution consists in scaling the QoS by changing the task period. In fact, Equation (1) shows that the bandwidth requested by a task decreases as its period increases. Using this solution, when a task detects that its reserved bandwidth is not sufficient to guarantee the current QoS, it can decrease the requested bandwidth by enlarging its period. When the task detects that the reserved bandwidth is enough, it can try to increase its QoS to a nominal desired value in order to attempt a better exploitation of the reserved resources.

The QoS adaptation scheme can work as follows: when the CBS scheduling error ϵ_i is greater than a specified threshold, the period is increased; whereas, when $\epsilon_i = 0$ for an interval of time larger than a specified amount, the period can be decreased. A lot of different and application-dependent strategies can be used for this purpose, so we identified a general period scaling algorithm that can be tuned to fit a lot of particular cases. If we define T_i as the desired activation period, and T_{act} as the actual period, a task can calculate its new period as

$$T'_{act} = (1 - \lambda_1 - \lambda_2)T_{act} + \lambda_1(\epsilon_i + T_{act}) + \lambda_2 T_i, \tag{4}$$

where λ_1 and λ_2 are two *forgetting factors* $\in [0, 1]$ such that $\lambda_1 + \lambda_2 < 1$.

The two factors λ_1 and λ_2 influence the speed with which the application decreases or increases its QoS. In particular, λ_1 controls the speed with which the application reacts to an increment in the scheduling error: when the task requests a bandwidth B_{req} bigger than the reserved bandwidth B_i, the application tries to change the task period to $T'_i = \frac{c_{i,j}}{B_i}$, using ϵ_i as an estimate of T'_i. If the period is increased too much, the factor $\lambda_2 T_i$ tries to decrease it in order to better use the reserved bandwidth.

It is worth noting that if $\epsilon_i = 0$, the actual period T_{act} tends to the desired activation period T_i. In fact, $\epsilon_i = 0 \Rightarrow T'_{act} = (1 - \lambda_2)T_{act} + \lambda_2 T_i$; since

$$\begin{array}{lcl} T_i < T_{act} & \Rightarrow & T_i < T'_{act} < T_{act} \\ T_i > T_{act} & \Rightarrow & T_{act} < T'_{act} < T_i \end{array}$$

this succession is limited and monotonic, then converges to a stationary value $\mathcal{T}$. In this case, $\mathcal{T} = (1 - \lambda_2)\mathcal{T} + \lambda_2 T_i$, hence $\lambda_2 \mathcal{T} = \lambda_2 T_i$, and the succession converges to T_i.

For some applications (such as video or audio players), it can be useful to select a discrete QoS model, in which not all period values can be selected for T_i, but only periods in $\mathcal{T}_i = \{T_{i_1}, T_{i_2}, \ldots, T_{i_n}\}$. In this case, the period adaptation algorithm can work as follows:

1. calculate T'_{act} as in Equation (4);
2. if $T'_{act} > T_{act}$, the new period is set at the value $T''_{act} = \max\{T \in \mathcal{T} : T \leq T'_{act}\}$;
3. if $T'_{act} < T_{act}$, the new period is set at the value $T''_{act} = \min\{T \in \mathcal{T} : T \geq T'_{act}\}$.

In this way the application also introduces an hysteresis in the period adaptation function, achieving a better stability.

3.2. Job skipping

Some applications may prefer to scale down their required bandwidth by not executing (skipping) some instances. In this case, Equation (1) becomes

$$B_i^{req} = \lim_{k\to\infty} \frac{\sum_{j=1}^{k} c_{i,j} h_{i,j}}{kT_i} \tag{5}$$

where

$$h_{i,j} = \begin{cases} 0 & \text{if job } J_{i,j} \text{ is skipped} \\ 1 & \text{otherwise} \end{cases}$$

In this case, the requested bandwidth is decreased by skipping instances when a task "is late". However, in some cases not all the frames can be skipped, and some tasks can be characterized by particular constraints on skipped jobs. For this reason, the decision of skipping or executing a job is left to the application.

The QoS adaptation can work as follows: when a job starts, the application checks whether the scheduling error ϵ_i is greater than a specified limit, and whether the job can be skipped. If these two conditions are verified, the job finishes immediately (with a virtually 0 execution time). Note that in this way an application does not reacts to overloads immediately, but skips jobs only after the overload has occurred.

Each application can define a different policy to decide whether a specified job $J_{i,j}$ can be skipped: in particular, if all the jobs can potentially be skipped the application can work properly also with a small reserved bandwidth. Otherwise, the application must define a function $s_i(j)$ indicating if $J_{i,j}$ can be skipped: in this case, the application needs at least a reserved bandwidth $B_i^{min} = \lim_{k\to\infty} \frac{\sum_{j=1}^{k} c_{i,j} s_i(j)}{kT_i}$.

4. Hierarchical QoS feedback control

In the previous section, we discussed how an application can adjust its QoS requirements to provide the best QoS under a fixed reserved bandwidth B_i. In this section, we show how such an approach can be nicely integrated with a bandwidth adaptation mechanism, such as the one presented in [2], to achieve a better performance.

Using an integrated solution, there are two *orthogonal* forms of adaptation:

- the reserved bandwidth adaptation realized by an active entity having a global system visibility, such as a QoS manager or the scheduler itself;
- the application dependent QoS adaptation, as presented in the previous section.

The integrated (hierarchical) approach presents the advantages of both methods, allowing the applications to scale their QoS when the bandwidth adaptation is not able to serve them properly. In fact, we will show that adaptive reservation can suffer when all the tasks require too much resources, and the QoS adaptation mechanism can solve this problem. On the other hand, the bandwidth adaptation mechanism allows applications to obtain the desired QoS without requiring any a-priori knowledge on their resource requirements.

4.1. Global bandwidth adaptation

As mentioned in the introduction, a global bandwidth adaptation mechanism is useful to prevent a misbehaved task from allocating too much resources for itself, so penalizing all the others.

In our previous work [2], we proposed a QoS manager having the visibility of all the tasks τ_i and their scheduling errors ϵ_i, that can use such a feedback information to allocate resources to minimize the scheduling errors according to some user-defined task importance values w_i.

The proposed adaptive bandwidth reservation mechanism is illustrated in Figure 1. It adjusts the reserved bandwidth $B_i(j+1)$ of the next job according to a *feedback function* $B_i(j+1) = f_i(B_i(j), \epsilon_i(j))$, where $B_i(j)$ is the bandwidth reserved to the current (j^{th}) job, and $\epsilon_i(j)$ is the scheduling error measured when $J_{i,j}$ finishes. If all tasks need to increase their reserved bandwidths "too much", violating Equation (2), the reserved bandwidths must be scaled down using a *compression equation* $B'_i = g(B, w)$, where $B = (B_1, B_2, \ldots, B_n)$ is the vector of the reserved bandwidths, and $w = (w_1, w_2, \ldots, w_n)$ is the vector of the tasks' weights.

The compression algorithm is the part of the bandwidth adaptation algorithm which requires global information: if

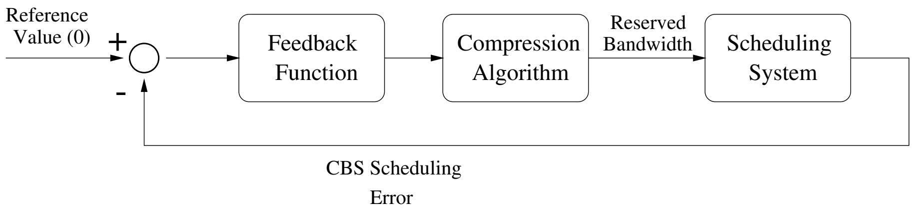

Figure 1. Bandwidth Adaptation Scheme.

$\sum_i B_i \leq B^{max}$ (where B^{max} is some desired maximum value such that $B^{max} \leq 1$), the bandwidth compression is not necessary and the adaptation is performed on a per-task basis. In this case, the tasks weights are not used. If $\sum_i B_i > B^{max}$, the assigned bandwidths have to be reduced (compressed) to maintain the system schedulable. In this case, it may happen that the bandwidth assigned to less important tasks is reduced too much, causing them to be in an overload condition, as expressed by Equation (3). In this situation, the local QoS adaptation mechanism may help to decrease the computational demand of the overloaded tasks, to achieve a graceful performance degradation.

4.2. Local QoS adaptation

We have shown that when the sum of the adapted bandwidths is greater than B^{max}, the less important tasks can suffer from local overloads. Indeed, the goal of the global adaptive reservation mechanism is to isolate task overruns in the less important tasks, independently from their requirements and periods. In this aspect, our approach differs from classical real-time techniques, in which task importance is inversely proportional to its period.

In this case, an overloaded task can try to scale down its requirements (by decreasing its QoS) as shown in Section 2. If such an adaptation is performed, the task may exit the overload condition, *reaching a lower QoS level in a controlled fashion*, otherwise the QoS degradation can be unpredictable.

If a task τ_i does not implement the local QoS adaptation, the less important tasks (the tasks τ_j with $w_j \leq w_i$) will be more penalized in terms of bandwidth, since the global reservation mechanism performs compression based on the importance values. Hence, the bandwidth of the less important tasks will be used to satisfy the QoS requirements of the most important tasks. Such a system behavior is consistent with our QoS model (avoiding overloads in the most important tasks). A possible concern can be that a misbehaved task having a high importance can compromise the QoS experienced by all the applications in the system. However, the importance w_i is assigned by the user, and can be used as a mechanism to penalize misbehaved tasks or applications that do not adapt their QoS properly.

Since the amount of resource requested by a task to provide a specified level of QoS is not always known (and only a feedback mechanism can be used to control the QoS) the global adaptive reservation mechanism alone may not be able to guarantee a minimum QoS to each task.

If an application dependent QoS adaptation is implemented, the task can scale down its resource requirements in order to provide a minimum QoS, if the task is guaranteed to receive a minimum amount of resources. For this reason, we modified the original adaptive reservations scheme in order to guarantee a minimum fraction of the CPU bandwidth to each task (note that if this minimum value is 0, the modified approach is equivalent to the original one).

4.3. Integrated Approach

To use our integrated QoS management approach, a new level of feedback has to be added to the feedback scheme of Figure 1, as shown in Figure 2. The inner loop controls the bandwidth B_i reserved by the global adaptive reservation, while the outer loop controls the bandwidth B_i^{req} requested by the application, using the local method. As explained above, the goal of the control loops is to obtain $B_i \geq B_i^{req}$. One of the major problems with this kind of hierarchy is that it can easily reach unstable conditions. For example, consider two tasks τ_1 and τ_2. By reacting to a transient overload, the global adaptive reservation mechanism can decrease B_1; if τ_1 reacts immediately by decreasing its QoS, when the transient overload finishes the bandwidth adaptation mechanism can increase B_2. In this way, τ_2 increases its QoS level, stealing bandwidth from τ_1, preventing it to recover its initial QoS level.

In order to solve this problem, we made the QoS adaptation action slower than the bandwidth adaptation one, so that QoS is changed only when the overload condition is long (in most cases, the QoS is not scaled in response to transient overloads).

5. Experimental results

In this section we report some significant experiments we performed on the HARTIK kernel [9] to show the advantages of QoS adaptation and the effectiveness of the method

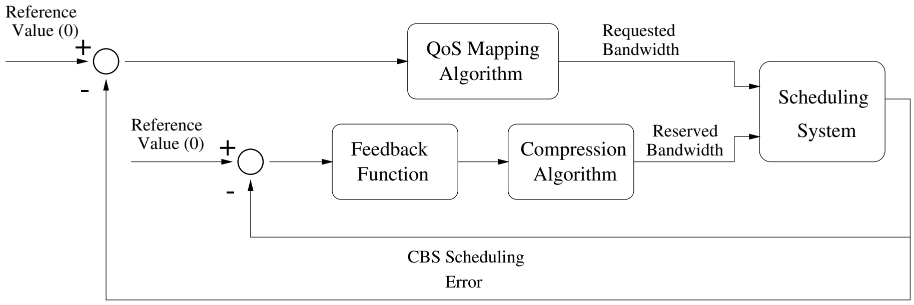

Figure 2. Two-Level Feedback.

proposed in this paper. In particular, we show that local (application dependent) QoS control and global (system wide) bandwidth adaptation can individually solve some problem, but only their integration can achieve better results without requiring any a-priori information on the task set.

5.1. Importance of local QoS adaptation

We start by showing that some form of QoS adaptation is needed for avoiding unpredictable behavior in the schedule.

In a first experiment, we ran four MPEG players decoding and visualizing movies concurrently. Each player executes as a task τ_i served by a CBS. In particular, we focus our attention on task τ_1, served by a CBS with parameters $(Q_1 = 8, T_1 = 30)$ and decoding a Variable Bit Rate (VBR) MPEG movie. The other tasks are used to create variations in the system workload. On a Cyrix P166+, the mean job execution time of τ_1 (that is, the mean frame decoding time) is $\overline{c_1} = 10ms$. Since the required bandwidth is $B_1^{req} = \frac{\overline{c_1}}{T_1} = \frac{10}{30} = 0.333 > B_1 = \frac{8}{30}$, the CBS serving τ_1 is clearly overloaded. As a consequence, if no adaptation mechanism is applied, task τ_1 will experience long unpredictable delays and its scheduling error ϵ_1 diverges towards infinity. Since the system workload is not stable (due to the variations in MPEG decoding times), τ_1 continuously increases (when some background time is idle and can be used by the task) and decreases its playback speed in an uncontrollable manner.

If τ_1 implements a skip strategy (skipping P and B frames when $\epsilon_1 > 0$), the mean job execution time decreases to $5ms$ and the required bandwidth is reduced to $B_1^{min} = \frac{5}{30} < B_1$. The scheduling error experienced by the task is shown in Figure 3. Using this technique, some frames are skipped (decreasing the QoS), but the movie is played at the correct rate (the requested one). Figure 4 shows the number of frames decoded by τ_1 as a function of time, with and without the skip strategy. From the figure it is possible to see that, using the skip strategy, the task can play frames at the correct rate, whereas if no adaptation strategy is implemented the task throughput is not constant, generating an unpleasant effect.

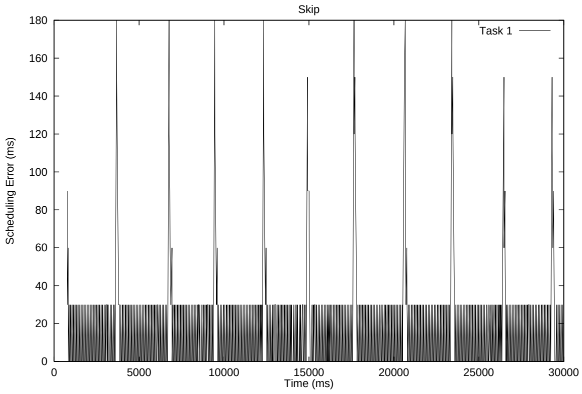

Figure 3. Scheduling error experienced by a task implementing a skip strategy.

When the server overload is higher, period adaptation can be more effective than skip for controlling the scheduling error. This fact has been verified in a second experiment, in which we ran three concurrent MPEG players. Each player executes as a task τ_i $(i \in \{1, 2, 3\})$ served by a CBS with parameters $(B_i = 0.3, T_i = 15)$, and τ_3 blocks for one second before restarting the entire sequence, to create a higher load variation.

In this experiment we also focus our attention on task τ_1, which decodes a VBR MPEG movie (different from the one used in the previous experiment) with mean job execution time $\overline{c_1} = 15ms$. Since the required bandwidth is $B_1^{req} = \frac{\overline{c_1}}{T_1} = \frac{15}{15} = 1 > B_1 = 0.3$, the server is clearly overloaded. As in the previous experiment, if no adaptation mechanism is applied, task τ_1 experiences unpredictable delays and its scheduling error ϵ_1 diverges towards infinity.

The effects of the uncontrolled scheduling error is shown in Figure 5, which plots the completion jitter (i.e., the interframe times $f_{i,j+1} - f_{i,j}$) as a function of time. The plot reveals some very high peaks (near to 1 second), corresponding to τ_3 re-activations.

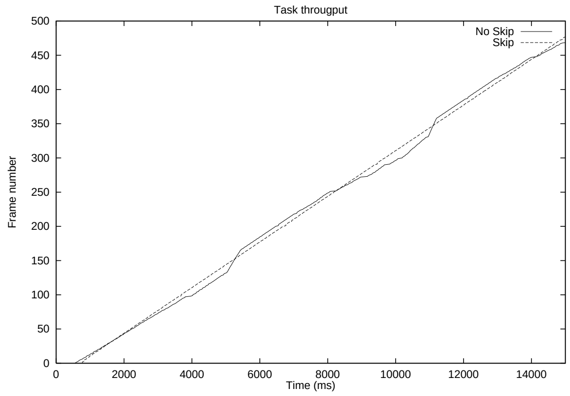

Figure 4. Comparison between skip and no skip.

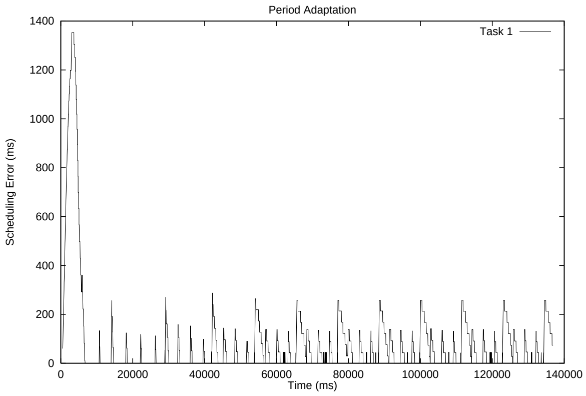

Figure 6. Scheduling error experienced by a task implementing period adaptation.

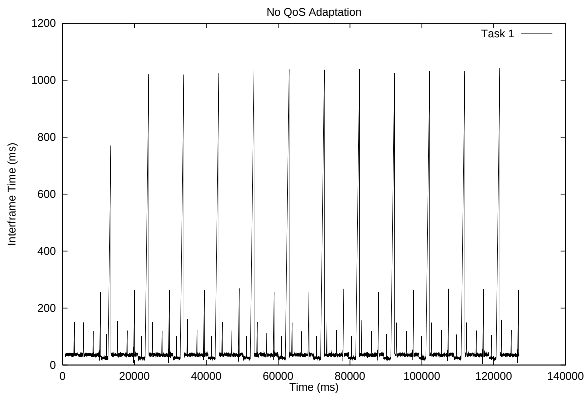

Figure 5. InterFrame times for a task not implementing QoS adaptation.

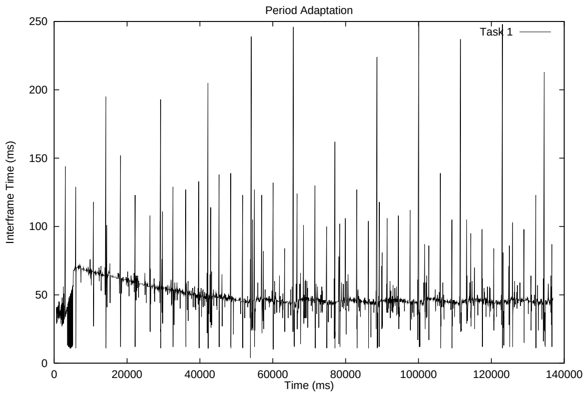

Figure 7. InterFrame times for a task implementing period adaptation.

When a QoS adaptation scheme is implemented by the player task, the scheduling error can be controlled, and the interframe times are more stable. In particular, we implemented the period adaptation rule expressed in Equation (4). The scheduling error achieved with this method is shown in Figure 6, whereas the interframe times are shown in Figure 7. The reader can see that some jitter is still present (since the frame decoding times are not constant), but it is much smaller than that shown in Figure 5. Period adaptation reduces the completion jitter from $1sec$ to $250ms$.

5.2. Importance of global bandwidth adaptation

The local (application-dependent) QoS adaptation mechanism (performed through period adaptation or by job skipping) allows each task to control its QoS within the allocated bandwidth. However, if the bandwidths are not properly assigned to tasks, the method may not achieve the expected results. For example, if two tasks τ_1 and τ_2 are assigned bandwidths B_1 and B_2 such that $B_1 > B_1^{req}$ and $B_2 < B_2^{req}$, τ_2 would scale down its QoS, but τ_1 would use less than its reserved bandwidth. This example suggests that, if τ_2 uses the bandwidth $(B_1 - B_1^{req})$ unused by τ_1, it would obtain a better QoS.

In order to show this fact, we ran an experiment with two tasks τ_1 and τ_2, served by two CBSs with parameters $(B_1 = 0.5, T_1 = 30)$ and $(B_2 = 0.4, T_2 = 30)$, decoding two different MPEG streams and implementing a period adaptation strategy. Figure 8 shows the interframe times of the two tasks. We can see that τ_1 increases its period, while τ_2 maintains it constant. This happens because the first MPEG video requires more than the reserved bandwidth to be played at the nominal rate, while the second movie requires less execution time. If we use an adaptive bandwidth reservation mechanism (see Section 4.1) to serve the two tasks (without any form of local QoS adaptation), both tasks can be served at their nominal rates, as shown in Figure 9. In this case, the adaptive bandwidth reservation mechanism provides better performance than the application dependent adaptation.

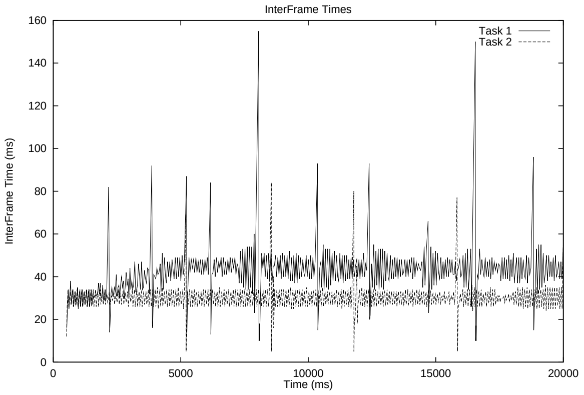

Figure 8. InterFrame times for two tasks implementing period adaptation with fixed reservations.

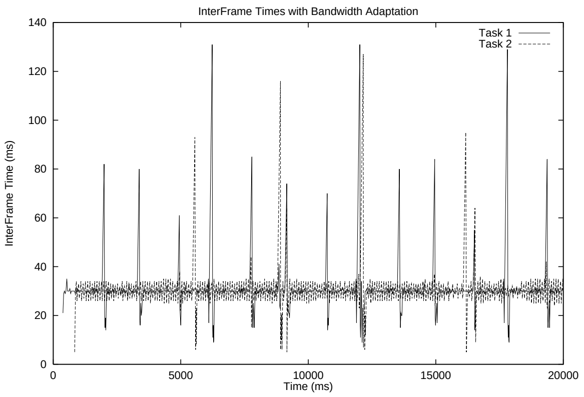

Figure 9. InterFrame times for two tasks served by adaptive bandwidth reservations.

However, adaptive bandwidth reservations can suffer if the sum of the required bandwidths is greater than 1 (and the compression mechanism is consequently used), and tasks do not implement any form of QoS adaptation. This is clearly illustrated in Figure 10, which plots the scheduling errors experienced by two MPEG players running on two tasks, τ_1 and τ_2, served under the global adaptive bandwidth reservation mechanism, where the sum of the requested bandwidths is greater than 1. In this case, since the system is overloaded, both scheduling errors increase to infinity. In the next section, we will see that a hierarchical QoS adaptation mechanism can solve this problem.

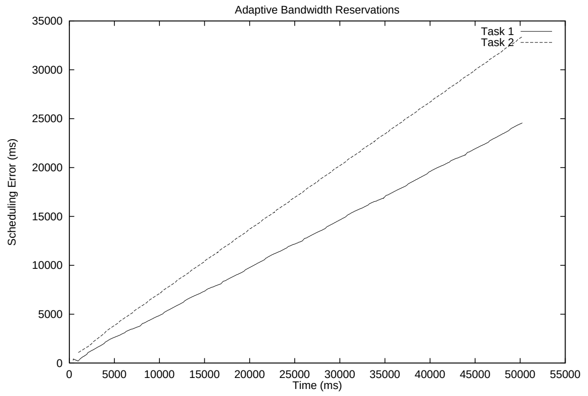

Figure 10. Scheduling errors of two tasks served by adaptive bandwidth reservation in an overloaded system.

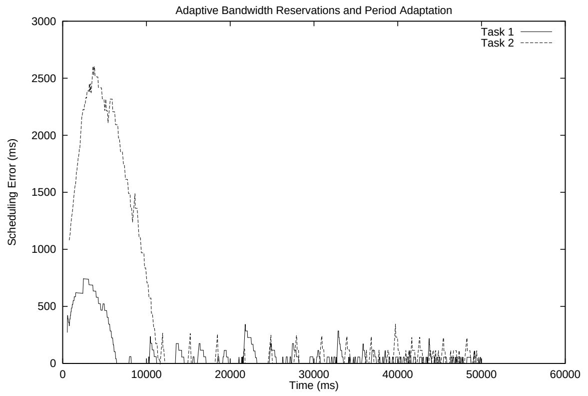

Figure 11. Scheduling errors of two tasks executing under a hierarchical QoS management.

5.3. Hierarchical QoS management

In the previous experiments we showed that, in some cases, an application dependent adaptation mechanism alone may be able to achieve the desired QoS level. In other cases, a fixed bandwidth reservation can penalize the system performance and the adaptive bandwidth reservation mechanism permits to increase the QoS experienced by a task. However, there are cases in which the bandwidth adaptation mechanism alone cannot succeed (for example, when the sum of the required bandwidths is greater than 1). In this cases, the two mechanisms can be combined, as shown in Section 4.3, to obtain a hierarchical QoS management.

To prove the effectiveness of the hierarchical control mechanism we ran an experiment in which the global bandwidth adaptation acts together with the application dependent mechanism (performed through period adaptation). Two tasks τ_1 and τ_2 (served by adaptive reservations), decode two MPEG videos and implement period adaptation. The results are reported in Figure 11, which plots the scheduling errors experienced by the tasks. As the reader can see, the combination of the two adaptation mechanisms is more effective and allows to reduce the scheduling error to zero. Moreover, it worth noting that no a priori knowledge is required from the task set to apply the proposed methodology.

6. Conclusions

In this paper we motivated the importance of implementing an application dependent QoS management scheme in time sensitive applications. We showed that when adaptation techniques are not used the resulting QoS can be highly unpredictable, and we proposed some simple techniques to be used together with a resource reservation strategy. In particular, we proposed and tested some solutions based on period adaptation and job skipping. Solutions based on imprecise computation can also be used to vary the computational demand of application tasks.

After motivating the importance of application dependent QoS management, we showed how this method can be integrated with adaptive resource reservations [2] to realize a *hierarchical QoS management* scheme which combines the advantages of the two approaches.

References

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

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

[3] L. Abeni, G. Lipari, and G. Buttazzo. Constant bandwidth vs proportional share resource allocation. In *Proceedings of the IEEE International Conference on Mutimedia Computing and Systems*, Florence, Italy, June 1999.

[4] D. Aparah. Adaptive resource management in a multimedia operating system. In *Proceedings of the 8th International Workshop on Network and Operating System Support for Digital Audio and Video*, Cambridge, UK, July 1998.

[5] S. Brandt, G. Nutt, T. Berk, and J. Mankovich. A dynamic quality of service middleware agent for mediating application resource usage. In *Proceedings of the IEEE Real Time Systems Symposium*, December 1998.

[6] H. Chu and K. Nahrstedt. CPU service classes for multimedia applications. In *Proceedings of the IEEE International Conference on Mutimedia Computing and Systems*, Florence, Italy, June 1999.

[7] C. L. Compton and D. L. Tennenhouse. Collaborative load shedding for media-based applications. In *Proceedings of the International Conference on Multimedia Computing and Systems*, 1994.

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

[9] G. Lamastra, G. Lipari, G. Buttazzo, A. Casile, and F. Conticelli. Hartik 3.0: A portable system for developing real-time applications. In *Proceedings of the IEEE Conference on Real-Time Computing Systems and Applications*, October 1997.

[10] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen. A scalable solution to the multi-resource qos problem. In *Proceedings of the IEEE Real Time Systems Symposium*, Phoenix, Arizona, December 1999.

[11] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Area on Communications*, June 1997.

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

[13] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Design and evaluation of a feedback control EDF scheduling algorithm. In *Proceedings of the IEEE Real Time Systems Symposium*, Phoenix, Arizona, December 1999.

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

[15] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.

[16] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for qos management. In *Proceedings of the IEEE Real Time Systems Symposium*, 1997.

[17] D. Rosu, K. Schwan, and S. Yalamanchili. Fara - a framework for adaptive resource allocation in complex real-time systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Denver, Colorado, June 1998.

[18] C. A. Waldspurger and W. E. Weihl. Stride scheduling: Deterministic proportional-share resource mangement. Technical Report MIT/LCS/TM-528, Massachusetts Institute of Technology, June 1995.

[19] D. K. Y. Yau and S. S. Lam. Adaptive rate controlled scheduling for multimedia applications. *IEEE/ACM Transactions on Networking*, August 1997.