

Legacy Real-Time Applications in a Reservation-Based System*

Luigi Palopoli, Luca Abeni

University of Trento

palopoli@disi.unitn.it, luca.abeni@unitn.it

Abstract

A remarkable research activity has been carried out in the past few years to support real-time applications with appropriate scheduling solutions. Unfortunately, most of such techniques can be used only if real-time applications use a specialised API, and if some important information (such as the Worst-Case Execution-Time) are known a-priori. In this paper, we present a novel technique, the Legacy Feedback Scheduling (LFS), for a class of legacy applications that need the support of a real-time scheduler but are not written using a specialised API and have unknown or varying execution requirements. The approach is based on the combination of a resource reservation scheduler and a feedback-based adaptation mechanism for identifying the correct scheduling parameters.

*The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7 under grant agreement nIST-2008-224428 "CHAT - Control of Heterogeneous Automation Systems"

1 Introduction

In recent years real-time computing has gained an increasing popularity, migrating from the small, albeit important, niche of embedded controllers to a much wider class of applications (e.g., multimedia and real-time communications). An important consequence of this paradigm shift is that general purpose operating systems (GPOSs) or their variants are making inroads into the realm of real-time applications, which are frequently executed on personal computers or portable devices. Illustrative of this trend are the modifications introduced into the GNU/Linux operating system to support embedded hardware, to reduce its footprint [18], and to decrease the latency¹ experienced by real-time applications [20].

This way a plethora of software components and applications developed for the GPOS can be used also on embedded devices. The problem developers have to face in using existing code for real-time applications is that traditional “legacy” applications are rarely developed with real-time constraints in mind. Even when temporal constraints are implicitly considered, such legacy applications have been typically developed without relying on any specialised real-time API.

In some of the traditional GPOSs, the responsiveness of the tasks identified as interactive is improved by increasing their priority. For example, the Linux scheduler contains some quite sophisticated mechanisms devoted to this purpose. Unfortunately, these heuristics do not offer satisfactory performance when applied to real-time applications.

The real-time scheduling theory provides the mathematical analysis techniques to guarantee that a set of applications can respect their temporal constraints, when scheduled with appropriate fixed or dynamic priorities [16]. These analysis techniques require the knowledge of such parameters as the Worst-Case

¹The latency experienced by a real-time task can be considered as a measure of the difference between the schedule generated by a real kernel and the “theoretical” schedule expected by the real-time theory.

Execution-Time (WCET) and minimum inter-arrival time, which are generally not known for legacy applications. In case of partial or imprecise knowledge on these parameters, the temporal guarantees can be entirely dissipated. This problem is addressed in reservation-based schedulers by “enforcing” upper bounds on the CPU guaranteed to each task through a mechanism similar to the token bucket. In other words, the mechanism “reshapes” the execution requests of a task so that it behaves as a task with a known WCET. In this way, if a task executes for more than expected, it does not affect the progress of the other tasks. This solution has often been applied to multimedia systems, and more recently to control applications [15] as well.

While reservation-based schedulers enable the use of real-time analysis to provide guarantees on the amount of time allocated to each application, they do not *directly* provide guarantees about application-level Quality of Service (QoS). To this end, it is possible to apply design methodologies [2] that guide the choice of the scheduling parameters to offer differentiated deterministic or stochastic QoS levels for each application. However, once again, these methodologies require an a-priori knowledge of the application requirements (e.g., the probability distributions of the computation time). Hence, this is hardly a viable solution for legacy applications. This problem can be addressed by applying feedback control to real-time scheduling [17, 14, 3, 10, 4, 23, 12]: while the applications execute, their real-time behaviour is monitored and corrective actions are taken changing the scheduling parameters so that specified QoS objectives are met. In this way, not only can execution requirements of unknown application be identified self-tuning the scheduling parameters, but it is also possible to change the resource allocation in time to accommodate the time-varying execution requirements of data dependent applications. The feedback-based adjustment of the scheduling parameters can be done by using either the classical PID controller [3, 10, 14] or nonlinear control schemes based on a prediction of the next

computation time [4]. In both cases, a fundamental assumption required to apply these approaches is that the application is structured as a sequence of real-time jobs (each one activated at some time and associated with a deadline). The use of a specialised API [24, 22] allows the system to measure the scheduling delays (upon the activation or termination of a job) and adjust the scheduling parameter using the feedback scheme. In [23, 12] the adaptation is made using different parameters (i.e., the real-rate) but the approach is still invasive on the application.

In this work, we take a different avenue: applying the feedback scheduling paradigm to a class of legacy applications in which implicit timing constraints *are not* exposed to the system by using a specialised API. This idea bears some resemblance with the one presented in [7, 8], where the authors adapt the heuristics of the Linux Kernel to real-time applications. A significant departure from this approach, though, is that we build our construction on the top of real-time scheduling solution rather than on a modification of an existing general purpose scheduler. In our preliminary work [6] (which is extended and subsumed by the present work), we have shown a framework in which the bandwidth assigned to the different tasks is used as an actuator in a feedback function collecting a much coarser “sensor” information than in traditional feedback schedulers (and in particular in the adaptive reservations [3, 4]). However, the “punctual” performance (intended as the ability to respect every single timing constraint) is necessarily lower. In this paper, we push much further this approach focusing on the control design issues. Namely, we formalise a general purpose controller (called LFSg), which in [6] was simply hinted at, and propose a completely new controller (called LFSp), whose design is particularly crafted for periodic tasks. Clearly, LFSp has a much better performance than LFSg when the task is applied on is indeed periodic. The performance gain with respect to LFSg is paid in terms of complexity and generality.

The rest of the paper is organised as follows. In Section 2, we offer some background on the real-time task model, the use of dedicated APIs that reflect it, and discuss its convenience. In Section 3, we provide some essential background on the scheduling algorithm used as a basis for our approach, and we show how it is possible to adapt the reservation parameters in legacy applications, even if they do not comply with the real-time task model. In Section 4, we propose two control schemes (LFSg and LFSp) for the legacy feedback scheduler introduced in Section 3. In Section 5, we show experiments obtained with an implementation of this technique in the Linux kernel and evaluate the performance of the control algorithms. Finally, in Section 6 we offer our conclusions and outline future work directions.

2 The Importance of Being Real-Time

The model used by the real-time scheduling theory considers a system \mathcal{S} composed by a set of concurrent activities (tasks): $\mathcal{S} = \{\tau_1, \tau_2, \dots, \tau_n\}$. The term *task* is used to denote either a process (owning a private memory space) or a thread (sharing the memory space with other threads). Each task τ_i consists of a stream of activities called *jobs*, $J_{i,1}, J_{i,2}, \dots$. Job $J_{i,j}$ is activated (becoming ready for execution) at time $r_{i,j}$, and finishes at time $f_{i,j}$ after executing for a time $c_{i,j}$. Moreover, $J_{i,j}$ is assigned a deadline $d_{i,j}$. The deadline is an execution constraint on the finishing time $f_{i,j}$.

Another important classification is on the activation pattern of the task. In particular, a real-time task τ_i is said *periodic* if $\forall j, r_{i,j+1} = r_{i,j} + P_i$, where P_i is the task's period. Most real-time operating systems (or real-time extensions of a GPOS) allow the programmer to write an application adhering to this model by an appropriate API. For instance, a real-time periodic task can be programmed as shown in Figure 1. The real-time requirements can be specified in the initialisation phase or in the primitive that creates the task. After an initialisation phase (and after setting up a periodic activation timer if the

```

1 void *RTTask(void *arg)
2 {
3     <initialisation>;
4     <start periodic timer, if task is periodic>;
5     while (!finished) {
6         rt_job_wait();
7         <job body>;
8         rt_job_finish();
9     }
10 }

```

Figure 1. Typical structure of a real-time task.

task is periodic), the task enters a loop in which it cyclically is blocked waiting for the next activation event (`rt_job_wait()`). For a periodic task this event is the expiration of a timer. When the event arrives, the `<job body>` is executed and then the call to the primitive (`rt_job_finish()`) notifies the termination of the job to the Operating System or to the Middleware ².

The adoption of the *real-time task model* shown above permits the easy assignment of temporal constraints to activities, and to check if such temporal constraints are respected. In particular, a hard real-time system has to guarantee that $\forall(i, j), f_{i,j} \leq d_{i,j}$. Therefore, the system can raise a critical exception whenever a deadline is missed, and the occurrence of this event can be detected when `rt_job_finish` is not executed before the expiration of a timer. For soft real-time systems, the timing requirements have a different nature and are associated to the QoS experienced by the application. For instance, it is possible to quantify the QoS by the probability $Pr \{f_{i,j} \leq d_{i,j}\}$ of “missing” a deadline . In this case, the execution of the primitive shown above allows the system to collect statistics on the QoS experienced of the application and/or to take corrective actions based on a feedback scheme (as specified in the next section).

As a consequence, to take full advantage of real-time features an application has to be programmed

²In fact, the two function calls `rt_job_finish()` and `rt_job_wait()` could be collapsed into one but we distinguished them for the sake of clarity

using a specialised API, explicitly identifying jobs and communicating their terminations to the system. In the past, alternative techniques have been proposed to indirectly identify these events. In [5], the activation and the deactivation of a task (i.e., the insertion and the removal from the ready queue) have been proposed as a means to identify jobs (a job is considered to start when the task activates, and is considered to finish when the task blocks). We point out that this is only a heuristic that in some situations could fail, since a task can be blocked inside `<job body>` (for example, while waiting data from disk or from another device). Generally speaking, if an application does not use a specialised API it may be hard or impossible to identify jobs (and hence to apply the real-time task model suggested in this section). Still, even traditional applications can be characterised by some *implicit*³ temporal constraints. For such applications, called *legacy time-sensitive applications* (or *legacy applications* for shortness) in this paper, it is impossible to provide real and exact real-time support. In other words, it is impossible to provide “punctual” guarantees (either hard or soft) to the behaviour of each job simply because no such notion is exposed by the application to the system. In this paper, we will try and offer some kind of “macroscopic” real-time support, putting aside the classical real-time task model.

3 Feedback for Legacy Applications

The scheduling technique for legacy applications that we propose in this paper is based on a paradigm known as the Resource Reservations.

A Resource Reservation [19, 22] is an abstraction that associates to each task τ_i a pair (Q_i, T_i^s) , meaning that task τ_i is reserved a time Q_i in each period T_i^s . The ratio $B_i = \frac{Q_i}{T_i^s}$ is called bandwidth and can be thought of as the “fraction” of CPU reserved to the task. The particular implementation of this

³These temporal constraints are called implicit because they are not explicitly declared through a specialised API.

- When τ_i is created, q_i and d_i^s are initialised to 0;
- When τ_i is activated at time t , if $q_i > (d_i^s - t)U_i$ then $d_i^s = t + T_i^s$ and $q_i = Q_i$, otherwise the scheduling deadline d_i^s and the budget q_i are left unchanged;
- While task τ_i executes, q_i is decreased as $dq_i = -dt$ (**accounting rule**);
- When the budget is exhausted ($q_i = 0$), it is recharged to Q_i and the scheduling deadline is postponed ($d_i^s = d_i^s + T_i^s$) (**enforcement rule**)

Figure 2. The CBS scheduler rules

paradigm that we will use in this paper is the Constant Bandwidth Server (CBS) algorithm [2]. The CBS implements the resource reservation paradigm using dynamic real-time priorities. Namely, each task (or group of tasks) is scheduled through a server, which is associated with a scheduling deadline d_i^s used to decide the (dynamic) priority of the task according to an Earliest Deadline First (EDF) paradigm.

The rules whereby the CBS assigns and manages the scheduling deadlines are reported in Figure 2. Whenever τ_i executes for a time Q_i , the scheduling deadline is postponed and the budget is recharged. In this way, a task that has consumed its budget in a reference server period $[kT_i^s, (k+1)T_i^s]$, has its priority lowered and holds the CPU only if no other task has pending execution requirements (associated with a shorter deadline). This mechanism guarantees that each task receives a fraction $B_i = Q_i/T_i^s$ of the CPU inasmuch as the schedulability condition

$$\sum_i B_i = \sum_i \frac{Q_i}{T_i^s} \leq 1 \quad (1)$$

holds (see [2] for more).

3.1 The legacy feedback scheduler

The CBS scheduler has been used in a variety of different contexts. Relevant to the context of this paper is the approach named “adaptive reservations” [3, 4]. The underlying idea is that the budget Q_i can be changed on a job-by-job basis using a feedback controller that “senses” the timing of the task w.r.t. the deadline at the end of the job.

Unfortunately, this scheme is not applicable to the case of legacy applications. Indeed, the timing of the task cannot be observed at the end of each job for the simple reason that a legacy application is not necessarily structured as a sequence of jobs (as in Figure 1). Even if the legacy application were structured in this way, we cannot assume that it notifies the start and the termination of each job by means of a standard system call.

In order to construct an adaptive scheme for real-time scheduling of legacy application, we can still use the budget Q_i as an actuation mechanism to steer the evolution of the system. However, both the sampling mechanism and the quantity measured for the feedback scheme (i.e., the “sensor” used to observe the evolution of the system) have necessarily to be very different from those used in the adaptive reservations. A very useful starting point is the following property:

Property 1 *Consider a CBS server used to schedule a task τ_i . Let d_i^s be the scheduling deadline of the server. Define $W_i(t_0, t)$ to be the total time executed by the task in the interval $[t_0, t]$ and $S_i(t_0, t)$ to be the computation time reserved by the CBS to the task in the same interval. Let $\epsilon_i(t) = d_i^s - t$ be defined as the difference between the scheduling deadline and the current instant of time. the following statements hold true:*

- *if $S_i(t_0, t) < W_i(t_1, t)$ then $\epsilon_i(t) > T_i^s$.*

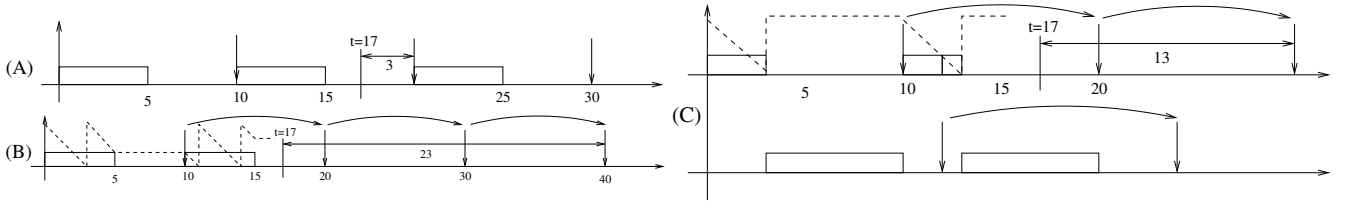


Figure 3. Examples of computation of the scheduling error $\epsilon(t)$.

- if $\forall t_1$, with $t_1 \in [t_0, t]$, we have $S_i(t_1, t) \geq W_i(t_1, t)$ then $\epsilon_i(t) \leq T_i^s$,

This property can easily be proved using the rules of the CBS algorithm. In this context, we just show some examples. In Figure 3.(A), we consider a task that periodically executes a function requiring 5 units of execution time every 10 time units. The task is scheduled by a CBS with $Q_1 = 6$ and $T_1^s = 10$. If we sample at time 17, we can see that $\epsilon_1(17) = 3 \leq T_1^s$ and we can conclude that the task received enough computation time (in fact more than needed). On the contrary, if, for the same task we choose $Q_1 = 3$, we get the evolution shown in Figure 3.(B). In this case, sampling ϵ_1 at time 17 we get $\epsilon_i(t) = 23 > T_1^s$ meaning that the task is suffering delays in its progress. One may be tempted to think that the quantity $\epsilon_1(t)$ is somehow related to the amount of backloged computation time for τ_1 awaiting execution. Unfortunately, this is not the case: for instance, if there are is another task in the system, in the same scenario considered in Figure 3.(B), we can get $\epsilon_1(17) = 13$, as shown in Figure 3.(C). In other words, for the same task and for the same allocation of CPU, we can get different values for $\epsilon_1(t)$ depending on the workload of the system. However, whatever the number of tasks presented in the system, for task τ_1 we would always get $\epsilon_1(17) > T_1^s$. In other words, by measuring $\epsilon_i(t)$, we simply get a binary information (task reserved too much CPU time or not).

As far as the sampling mechanism is concerned, in the absence of any particular restriction on *when* the quantity ϵ_i can be observed, we propose, in this paper, the classical periodic sampling, which is a commonplace solution in digital control applications. Thereby, the *Legacy Feedback Scheduler* – **LFS**

in the rest of the paper – can be organised as follows:

1. At time kT^{sample} , with $k \in \{0, 1, 2, \dots\}$ the scheduling error of each time sensitive task τ_i is sampled producing the vector $\epsilon = [\epsilon_1, \dots, \epsilon_n]$;
2. The vector of the CBS budgets $Q = [Q_1, \dots, Q_n]$ is computed by the feedback controller $Q = f(Q, \epsilon)$; the budgets are used the scheduler in the interval $[kT^{sample}, (k+1)T^{sample}]$.

In the next section, we will illustrate alternative design options for control design.

4 Control design

Based on the discussion above, a task system is a multiple input multiple output (MIMO) plant, whose inputs are the budgets and whose outputs are the binary sensors described above.

A useful intuition is to think of the system as a collection of tanks, in which a fluid flows in through an inlet (computation time) and flows out through an outlet (the budget Q_i reserved to the task). Our control goal is to maintain the level of the fluid inside the tanks very close to a reference value. Our sensors are only able to say if the fluid is above or below the chosen level but do not measure the amount of the deviation. The quantity of fluid flowing out from the system is constrained by Equation (1).

This control task can be logically partitioned into two different sub-tasks: 1) estimating the computation request of each task, 2) finding a good balance between the time that can actually be granted to each task without violating the Constraint (1). To address the second issue we use a simple scheme that minimises the distance between the desired allocation and the attained one. Namely, if at time t , the budget required for each task τ_i is $Q_i^{req}(t)$ the request can be granted if the requested bandwidth respects Condition (1). If it does not, we grant each task i a non-negative budget $Q_i(t)$ that minimises the maximum

distance from the request $\min_{Q_i(t)} \{ \max_{i=1, \dots, N} \{ w_i | Q_i(t) - Q_i^{req}(t) | \} \}$ subject to the constraint given by Condition (1). As shown in [21], this is a linear programming problem that can be solved in closed form with the only computation burden of reordering an array whose size is equal to the task number. For instance, if we consider two tasks and if each of the budget request is lower than the respective server period T_i^s , the budget allocated to the task is given by:

$$Q_i(t) = \begin{cases} Q_i^{req}(t) & \text{if } \sum_{j=1}^2 \frac{Q_j^{req}(t)}{T_j^s} \leq 1 \\ Q_i^{req}(t) - \frac{\sum_{j=1}^2 \frac{Q_j^{req}(t)}{T_j^s} - 1}{w_i \sum_{j=1}^2 \frac{1}{w_j}} & \text{otherwise} \end{cases} \quad (2)$$

where w_i is the relative weight of task τ_i , and j is an index running over the task set. With this solution, if the weight w_i is very large, then task i receives a budget very close to its request. If on the contrary it is very small (compared to the other weights) the task will receive a budget $Q_i(t) = T_i^s (1 - \sum_{j \neq i} \frac{Q_j^{req}}{T_j^s})$, i.e., what is left after the other task satisfies its request.

As far as the problem of estimating the computation request of each task is concerned, we have developed two different solutions. The first one – call this solution Legacy Feedback Schedule general (LFSg) – applies to very general applications. This solution is inspired by the congestion control mechanism used in the TCP protocol [13]. It makes no assumption on the type of considered application, its main requirement being the generality. However, its performance is necessarily not optimal, and in general better solutions can be found for specific classes of application. Indeed, the second algorithm that we propose – Legacy Feedback Schedule periodic (LFSp) – applies to periodic tasks and it assumes that one has a prior knowledge on the task period. Because of the additional information it uses, its performance is obviously better than that of LFSg.

The simulations described in this section can be reproduced by downloading the simulator we used from the website <http://www.disi.unitn.it/~palopoli/LFSsim.tgz>.

4.1 LFSg

The algorithm starts from an initial guess of the computation requirements of the task. When a new sample arrives at time kT^{sample} , if the sensor reading is $\epsilon_i(kT^{sample}) > T_i^s$ then the computation time is increased by multiplying the previous value by a constant, otherwise it is decreased by subtracting a constant:

$$Q_i^{req}(h+1) = \begin{cases} aQ_i(h) & \text{if } \epsilon_i(hT^{sample}) > T_i^s \\ \max\{Q_i(h) - b, 0\} & \text{if } \epsilon_i(hT^{sample}) \leq T_i^s \end{cases} \quad (3)$$

The two parameters a and b are positive real numbers with $a > 1$. The algorithm combines a quick (multiplicative) increase mechanism to find an allocation that provides the task with enough resources, with a slower (subtractive) decreases mechanism that allows us to fine-tune the allocation.

Example 1 Consider a periodic task with period $T_1 = 20ms$ and assume that the computation time is a step function: $c_{1,h} = 8$ for $h \leq 30$ and $c_{1,h} = 10$ for $h > 30$. Suppose that the task is scheduled through a CBS whose period is equal to that of the task and whose budget is chosen applying the LFSg algorithm with period 480. In Figure 4, we report the simulation results for the task computation time and for the time allocated by the controller (normalised to the task period) for different values of a and b .

The example above suggests some considerations. First, quantisation in the sensor inhibits an exact regulation on the reference point. In fact, at the steady state the allocated time varies in a neighbourhood of the reference point. Very frequently the budget oscillates between a lower bound and an upper bound (a *limit cycle*). To evaluate the effect of parameter a , consider a step variation in which the computation time changes to a constant value \bar{c} . We define the rise time τ_r as the time required to the controller to change the value of the budget Q from an initial value Q_0 to a value that is greater than or equal to \bar{c} (assuming that its request is entirely granted by the compression mechanism). Since the budget is multiplied by

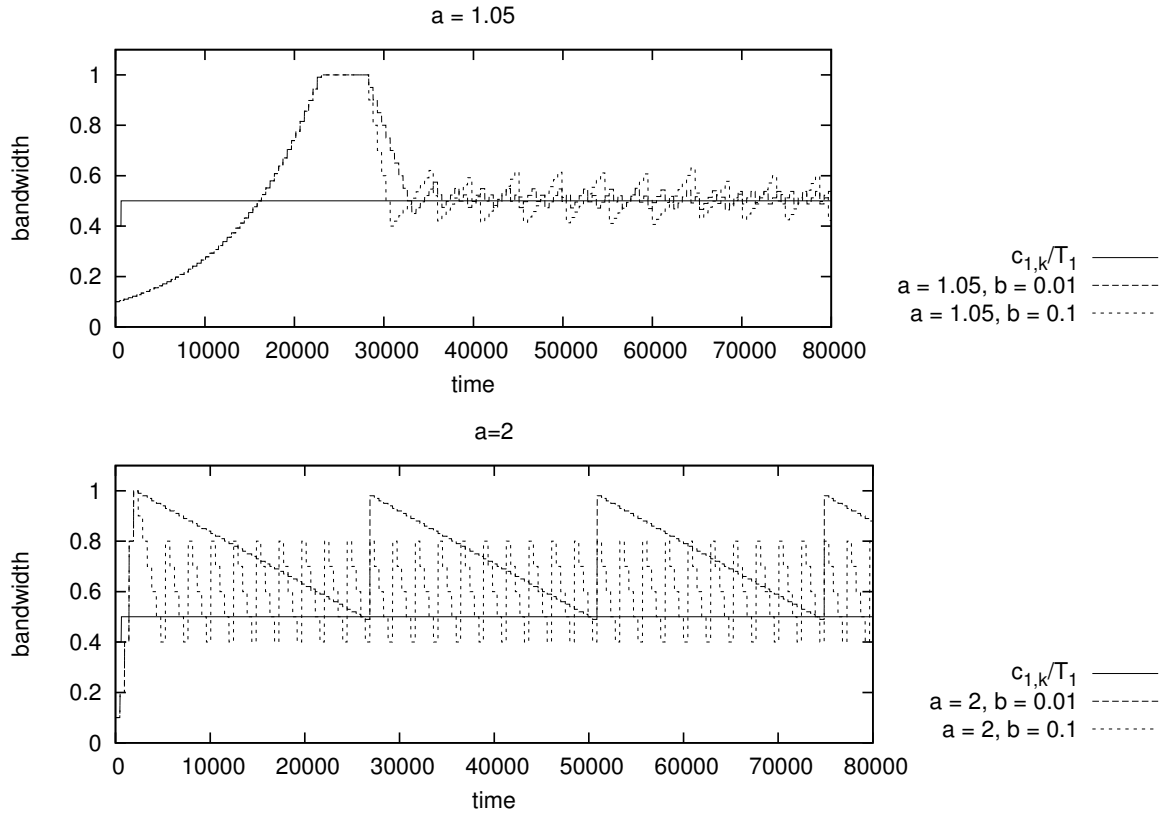


Figure 4. Step responses for Example 1 using LFSg

a at each sampling period, it is possible to estimate the rise time as $\tau_r \approx T^{sample} \left\lceil \frac{\log c - \log Q_0}{\log a} \right\rceil$. In our example, for $a = 1.05$ we get $\tau_r \approx 48 \cdot 480$ and for $a = 2$ we get $\tau_r \approx 4 \cdot 480$. On the other hand, the choice of a has an impact on the upper bound of the limit cycle. Indeed, in Figure 4 we can see that for $a = 2$ the system enters a limit cycle, with an upper bound \bar{l} that is approximately equal to 90% of the period, whereas for $a = 1.05$ the upper bound of the limit cycle is around 55% of the period (the desired value of Q is 50% of the period). Similar considerations apply to the choice of b , thereby it is possible to control the velocity in reducing the bandwidth after overcoming the regulation value. Higher values of b determine a quicker descent but also decrease the lower bound of the limit cycle. In the figure, choosing $b = 0.1$ determines a lower bound for the limit cycle around 0.4% of the period,

while the choice $b = 0.01$ leads to a lower bound 0.49% of the period. This discussion can be repeated in similar terms if we consider a case in which the computation time oscillates around an average value. If the sampling time of the controller is long enough, the algorithm estimates the average computation time through the window and the discussion on the choice of the parameters can be rephrased referring to the ability for the algorithm to identify this mean. This point is shown in the experiment section.

4.2 LFSp

For the reader's convenience, the pseudo-code of the LFSp algorithm is reported in the appendix. In this section, we provide a simplified explanation of the basic mechanisms.

4.2.1 Model

Assuming that a CBS having the same period of the task is used a scheduling mechanism, it is possible to write the following dynamic model:

$$W_{i,k+1} = \max\{0, W_{i,k} - Q_{i,k} + c_{i,k}\}, \quad (4)$$

where $W_{i,k}$ is quantity of backlogged computation time at the beginning of the k^{th} job, $Q_{i,k}$ and $c_{i,k}$ are respectively the budget reserved for the k^{th} job and the computation time of the k^{th} job. Clearly, we are not able to measure $W_{i,k}$. Based on the discussion in the previous section, we have a binary sensor $\text{sens}_i(t)$, such that $\text{sens}_i(t) = 0$ if $W_{i,k} = 0$ and $\text{sens}_i(k) = 1$ if $W_{i,k} > 0$. In the rest of the section we will remove the i subscript, since the discussion is referred to a single task τ_i .

In addition to the ‘‘periodic behaviour’’ assumption, LFSp has been developed assuming that: 1) the sampling period T^{sample} is an integer multiple of T_i^s ($T^{sample} = HT^s$), 2) the sampling operation is synchronised with the task periods (i.e., the first sample is taken upon a job arrival), 3) the computation

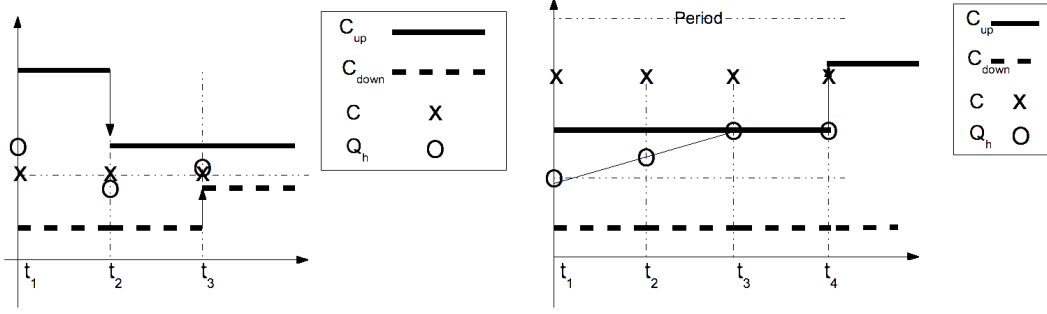


Figure 5. Left: The binary search used in the LFSp algorithm to identify the computation time. Right: Response to a step

time c_k is piecewise constant (i.e., it changes its values in some “breakpoints” and it remains constant elsewhere). As shown next, although developed under these assumptions, the algorithm has a good performance in a much larger range of situations.

4.2.2 The algorithm

The algorithm has to fulfil two different tasks: 1) identify the computation request of the task, when the computation time is constant, 2) react to changes in the computation time (which take the form of a “step”). In rest of the section, we discuss how we tackle each of the two tasks.

Constant computation time. In order to identify the computation request of a task whose computation time is equal to an unknown constant c , let us focus on the algorithm execution when it receives the sample $\text{sensor}(hT^{\text{sample}}) = \text{sensor}(hHT^s)$. For notational convenience, we will henceforth imply the multiplication by the sampling period (e.g., $\text{sensor}(h)$ will be used instead of $\text{sensor}(hT^{\text{sample}})$). Looking at the model in Equation (4), and taking into account that the budget $Q(h)$ decided by the controller will be held constant throughout the next sampling period ($Q_k = Q(h)$ for all $k \in [hH, (h + 1)H]$), we can

choose

$$Q^{req}(h) = \tilde{c}(h) + \frac{\tilde{W}(h)}{H}$$

where $\tilde{c}(h)$ is the estimate of the computation time, and $\tilde{W}(h) > W_{hH}$ is an upper bound estimate of the workload (recall that the actual value of the budget will be obtained by using Equation (2)). To construct the estimate $\tilde{c}(h)$, we start from a conservative estimation of a range $\tilde{c}^{up}(0)$, $\tilde{c}^{down}(0)$ such that $c \in [\tilde{c}^{down}(0), \tilde{c}^{up}(0)]$ and perform a binary search. Suppose that, $\text{sens}(h) = 0$. Under the given assumptions, we can easily conclude that during the $h - 1$ -th sampling period $Q(h - 1) \geq c$. Therefore, we can update $\tilde{c}^{up}(h) = Q(h - 1)$ (line 27 in the listing). On the contrary, if $\text{sens}(h) = 1$ and $\text{sens}(h - 1) = 0$, we can conclude that $Q(h - 1) < c$ and we update \tilde{c}^{down} : $\tilde{c}^{down}(h) = Q(h - 1)$. The estimate $\tilde{c}(h)$ is then constructed as $\tilde{c}(h) = (\tilde{c}^{up}(h) + \tilde{c}^{down}(h))/2$ (line 38 in the listing). An example execution of the search is displayed in Figure 5.(A).

As far as the computation of $\tilde{W}(h)$ is concerned, if $\text{sens}(h) = 0$, we have $W_{hH} = 0$ (line 16). Then we can choose $\tilde{W}(h) = 0$. If $\text{sens}(h) = 1$, we have $W_{hH} = W_{(h-1)H} + H(c - Q(h - 1))$. Hence, we can find $W_{hH} \leq \tilde{W}(h) = W_{(h-1)H} + H(c^{max}(h - 1) - Q(h - 1))$ (line 36).

Tracking step variations. It is easy to see that the binary search shown above converges if the computation time of the task always remains within the $[\tilde{c}_{down}(\cdot), \tilde{c}_{up}(\cdot)]$ interval. However, in response to a step in the computation time, this situation could no longer be satisfied. For instance, consider the situation in Figure 5.(B). Suppose that the algorithm has reached the tolerance width for the interval (just to simplify the discussion), hence it is no longer modified. At time t_1 the computation time steps out of the interval. After a first transition from 0 to 1, the sensor keeps returning 1 for several times in a row. The algorithm responds increasing the estimate of the computation time from the mid-point of the interval $[\tilde{c}_{up}, \tilde{c}_{down}]$ (in Figure 5.(B), in the sampling instant between t_1 and t_3 , and lines 48 to 55 in the

listing). The number of steps required to reach the upper bound is a configurable parameter (`len` in the listing). When the upper bound is reached, the algorithm keeps estimating the computation time on the upper bound until the estimated workload $\tilde{W}(h)$ reaches 0^4 . At this point, if the sensor still returns 1, it means that \tilde{c}_{up} is no longer a valid bound. Therefore it is increased to the midpoint between the current value of $Q(h)$ and the server period and the process continues (sampling instant t_4 in Figure 5.(B) and line 45 in the listing). The algorithm operates symmetrically if the computation time is reduced below c_{down} and the sensor keeps returning 0. It is possible to show that this procedure converges.

Example 2 In the example in Figure 6, we put on display the evolution of the allocated bandwidth for a system of two tasks using both the LFSp algorithm (with two different choices of the weights) and the LFSg algorithm (with $a = 1.2, b = 0.01$). As it is possible to see, with the LFSp algorithm we get a quicker response (i.e., a small rise-time), and a very small steady state error (in fact, it can be freely tuned operating with the tolerance parameter of the algorithm). In the example, around the time 10000 the computation time of the second task (continuous line) increases generating an overload condition. In this case, the weights play an important role: by choosing a larger weight for τ_2 , it receives a bandwidth sufficient to track the computation time change (with τ_1 paying the price), while if τ_1 has the larger weight the step on τ_2 is only partially tracked. A similar behaviour is observed in the *LFSg* algorithm. The resilience of the algorithm to the anomaly is lower, and recovering the equilibrium after the overload takes a longer time. This is due to the better estimation of the computation workload made by LFSp, which, pointed out above, uses more information than *LFSg*.

⁴If the computation request Q^{req} is not reduced by the compression function (2) this situation is obtained in one sampling period

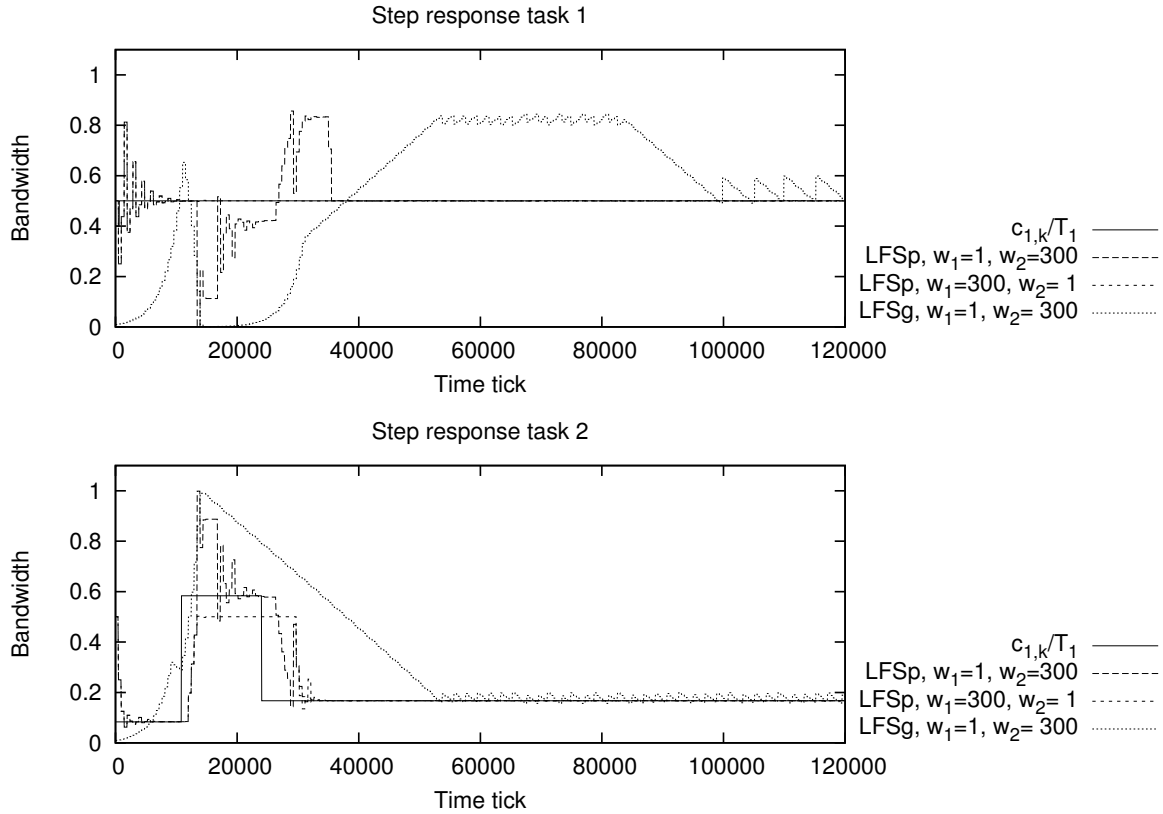


Figure 6. Piecewise constant computation time: evolution of the allocated bandwidth for two tasks using the LFSp and the LFSg algorithms for different choices of the weights.

5 Experimental Results

In this section we show the effectiveness of the presented solution in the context of a real implementation, where the system is affected by practical aspects that have not been considered in the earlier section and the computation time of the threads are not piecewise constant. In particular, we compared the performance of the LFSp and the LFSg on both a synthetic set of applications (whose workload profile is similar to that of multi-modal control applications) and on a real implementation of a video player.

To implement the mechanisms described in this paper we needed an implementation of the CBS

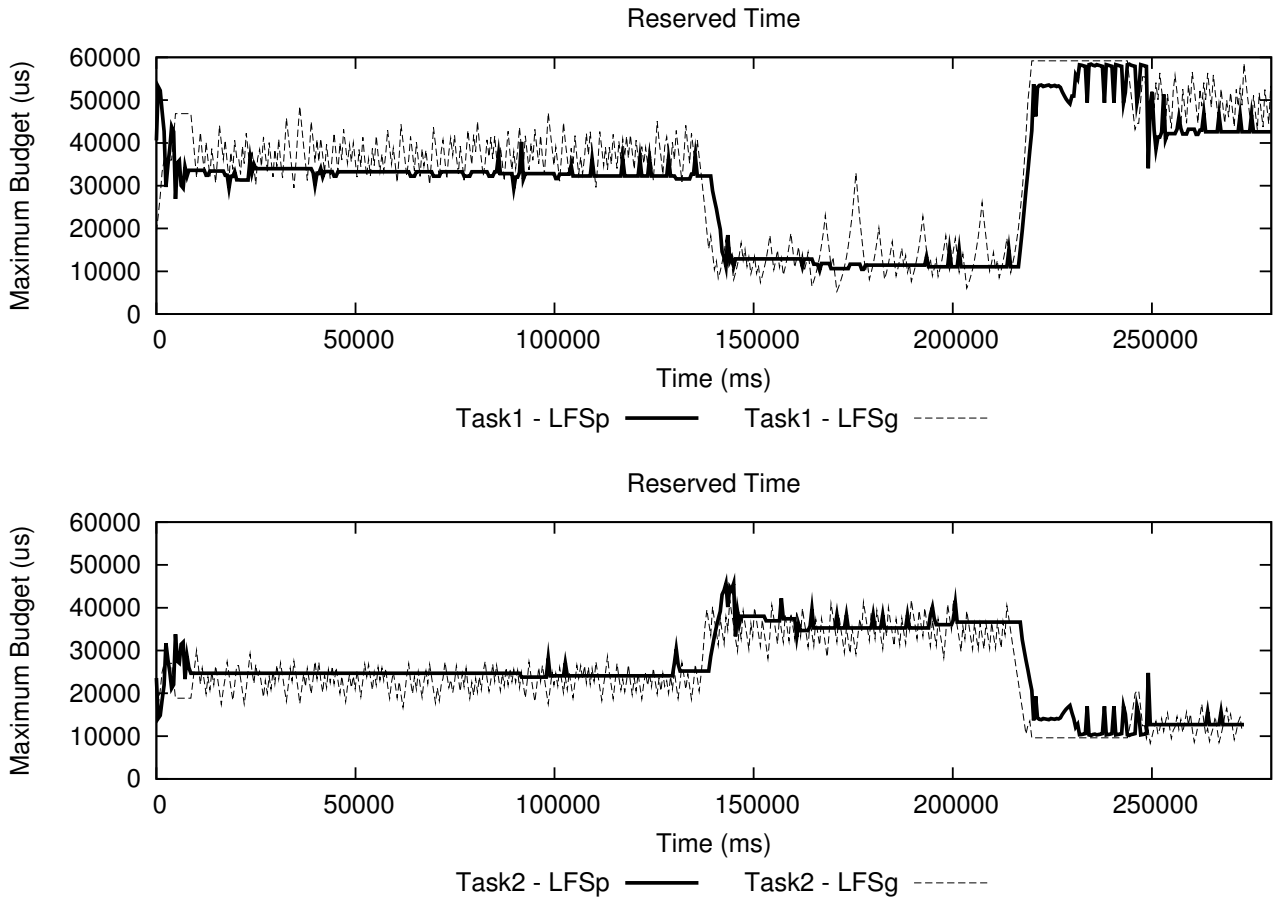


Figure 7. Evolution of the reserved time for two periodic tasks τ_1 and τ_2 .

scheduler, a “sensing” mechanism that allows us to measure the scheduling error, and a software component that is activated periodically and implements the control algorithm. As far as the first two points are concerned, we opted for an implementation of the CBS *inside* the Linux Kernel along the lines described in [5]. The feedback mechanism has been implemented in a middleware running in user space. A different possibility is to implement both the feedback controller and the CBS scheduler in user space. To this end, one can use a real-time scheduling API (such as the one featured by the POSIX1003b standard) as suggested in [11]. In this case, the user space module forces the scheduling decisions (following the CBS rule) by setting the maximum priority to the task chosen for execution.

All the experiments have been run on a PC endowed with an AMD64 CPU at 1.8Ghz and 512MB of RAM, running a modified version of the `2.6.21.4-rt12-cfs-v17` Linux kernel.

5.1 Synthetic workload

In the first test, we have considered two periodic tasks τ_1 and τ_2 (with periods $T_1 = 80ms$ and $T_2 = 60ms$) characterised by multiple working modes. For each mode, the task executed with a random computation uniformly distributed in a range. In particular, τ_1 has an execution time uniformly distributed in $(30ms, 40ms)$ when running in mode 1; in $(5ms, 15ms)$ when running in mode 2, and in $(45ms, 55ms)$ when running in mode 3. Task τ_2 , on the other hand, has execution times uniformly distributed in $(15ms, 25ms)$ (mode 1), $(30ms, 40ms)$ (mode 2), and $(10ms, 20ms)$ (mode 3). The two tasks run in mode 1 from time $0s$ to time $136s$, in mode 2 from time $136s$ to time $216s$, and in mode 3 after time $216s$. This behaviour was closely inspired by multi-modal control application (e.g., the ones used in avionics during the different phases of the flight): in each mode a specially designed feedback controller is executed with a computation time fluctuating in a small range around a mean value. Such fluctuation are typically due to data dependencies or architectural effects (e.g., cache misses, pipeline stalls, ...).

In the experiment, a CPU controller with period $480ms$ has been used to dynamically adapt the maximum budget Q of the two servers (the parameters $a = 1.2$ and $b = 0.1$ have been used for LFSg, and the two tasks have been assigned the same weight). The evolution of Q_1 and Q_2 is shown in Figure 7: as it is possible to see, after an initial transient the maximum budget is correctly adapted around the execution times of the two tasks. It is worth noting that the LFSp algorithm (taking advantage of a greater knowledge on the task periodicity) causes much smaller fluctuations in the reserved time than LFSg.

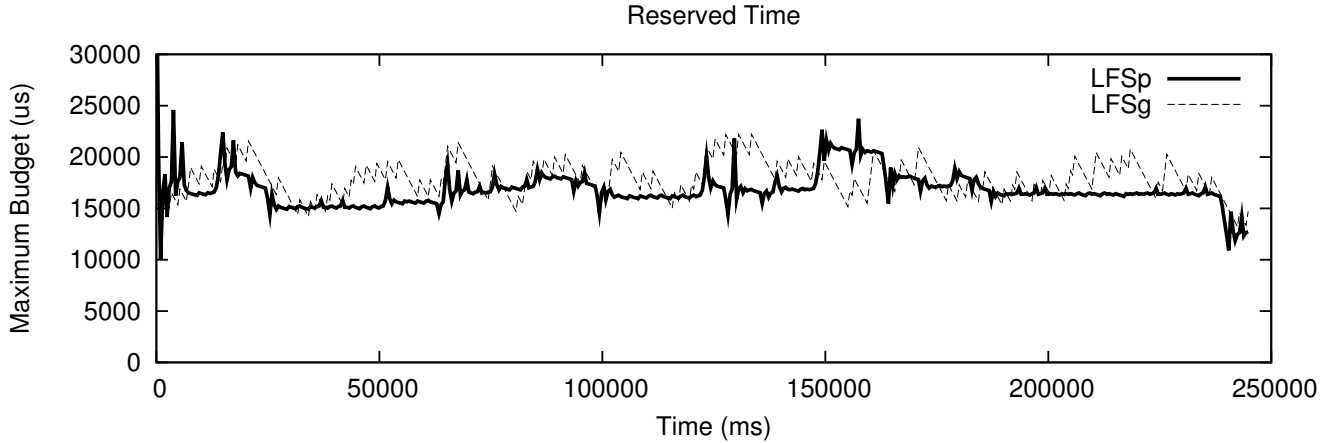


Figure 8. Reserved time for the video decoder.

This behaviour bears a close resemblance with the one we observed for piecewise constant computation times in Section 4.

5.2 Video Player

In a second set of experiments we have tested the control algorithms using a video player based on GTK [1] and on the FFMpeg [9] codecs. The player is a Linux application that reads compressed frames from disk, decodes them, and displays them at $25fps$ (hence, with a period $P = 40ms$); we decided to use a simplified custom implementation for the player (instead of an existing application like mplayer, totem, or vlc) to simplify the collection of statistics on such QoS metric as the inter-frame time (i.e., the time elapsed between the display of two consecutive frames).

The player has been tested on different MPEG2 videos measuring the inter-frame times denoted as if_k for the k^{th} activation. Since the video is played at $25fps$, ideally we would require an interframe time that is constant and equal to $40ms$. A good QoS metric is hence the experimental probability $Pr\{if_k > 40ms\}$ of experiencing an inter-frame time larger than $40ms$. In a first set of experiments,

we have used the video-player as a legacy application with the LFSp and LFSg adaptive schedulers. For the LFSg player we achieved $Pr\{if_k > 40ms\} = 13.074\%$. The results obtained for the LFSg were very similar (the difference being in a few decimal point). However, as shown in Figure 8, the bandwidth reserved by the LFSp is smaller. The frame decoding times are obviously variable, and the `top` command in this run indicated that the player consumes around 38%/40% of the CPU time (hence, the average frame decoding time is around 15ms or 16ms). As shown in the figure, the LFSp controller estimates a local average of the amount of time needed to properly serve the player, reducing the spikes in the CPU allocation and the overestimations/underestimations. Keeping as much as possible a constant bandwidth (without detrimental effects on the QoS) is a particularly important feature of the LFSp algorithm for real implementation. Indeed, when resource reservations are implemented in an OS kernel frequent (and un-needed) changes in the scheduling parameters can introduce a significant overhead and be more difficult to actuate when the Q parameter is quantised (as it needs to be for efficiency reasons).

In another set of experiments, we have used a different version of the player, written using a real-time API conforming to the scheme in Figure 1. In particular, we used the Adaptive Reservation scheme (based on a PI controller) shown in [3]. In this case, we reported $Pr\{if_k > 40ms\} = 1.562\%$. The impressive improvement requires, however, a profound change on the application, which is not generally possible. We point out, though, that the performance obtained with the LFSp and LFSg scheme is comparable to the one of a static allocation of resources only after a demanding sequence of trial and error iterations: the scheduler based on the legacy feedback displayed an evident self-tuning ability.

6 Conclusions and Future Work

In this paper, we have developed the Legacy feedback Scheduler, first introduced in our preliminary paper [6], putting the stress on the issue of control design. In particular, we have showed two approaches. The first one, LFSg, had been outlined in our previous work and is here described in detail. It is extremely simple and of general applicability. The second one, called LFSp, has been designed for periodic legacy applications and has a better performance than LFSg on this type of tasks.

The performance of the two LFS algorithms have been evaluated through a set of simulations and a set of experiments on a real implementation (based on Linux), proving the robustness of the algorithms and their applicability to very complex types of applications. The most important line for future work is to investigate on online algorithms that extract additional information about the task set to improve the effectiveness of the feedback action. A complementary research line is on the development of specially featured control algorithms for specific classes of applications (along the lines of the LFSp algorithm proposed in this period).

References

- [1] The gtk+ project. <http://www.gtk.org>.
- [2] 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.
- [3] 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.
- [4] L. Abeni, T. Cucinotta, G. Lipari, L. Marzario, and L. Palopoli. QoS management through adaptive reservations. *Real-Time Systems Journal*, 29(2-3), March 2005.

- [5] L. Abeni and G. Lipari. Implementing resource reservations in Linux. In *Proceedings of Fourth Real-Time Linux Workshop*, Boston, MA, December 2002.
- [6] L. Abeni and L. Palopoli. Adaptive real-time scheduling for legacy applications. In *IEEE International Conference on Emerging Technologies and Factory Automation, 2008 (ETFA08)*, Hamburg, September 2008.
- [7] S. A. Banachowski and S. A. Brandt. The best scheduler for integrated processing of best-effort and soft real-time processes. In *Multimedia Computing and Networking 2002 (MMCN02)*, San Jose, California, January 2002.
- [8] S. A. Banachowski and S. A. Brandt. Better real-time response for time-share scheduling. In *International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 2003)*, Nice, France, April 2003.
- [9] F. Bellard et al. Ffmpeg. <http://www.ffmpeg.org>.
- [10] G. T. 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.
- [11] H. Chu and K. Nahrstedt. A soft real-time scheduling server in UNIX operating system. *Lecture Notes in Computer Science*, 1309:153–162, 1997.
- [12] A. Goel, J. Walpole, and M. Shor. Real-rate scheduling. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004*, pages 434–441, 2004.
- [13] V. Jacobson. Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 25(1):157–187, 1995.
- [14] B. Li and K. Nahrstedt. A control-based middleware framework for quality-of-service adaptations. *IEEE Journal on Selected Areas in Communications*, 17(9):1632–1650, 1999.
- [15] Q. Ling and M. Lemmon. Soft real-time scheduling of networked control systems with dropouts governed by a markov chain. *American Control Conference, 2003. Proceedings of the 2003*, 6:4845–4850 vol.6, 4-6 June 2003.

- [16] C. L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1), 1973.
- [17] 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 20th IEEE Real-Time Systems Symposium*, Phoenix, AZ, December 1999.
- [18] M. Mackall. Linux-tiny and directions for small systems. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2004.
- [19] C. W. Mercer, R. Rajkumar, and H. Tokuda. Applying hard real-time technology to multimedia systems. In *Workshop on the Role of Real-Time in Multimedia/Interactive Computing System*, 1993.
- [20] I. Molnar et al. Real-time Linux wiki. <http://rt.wiki.kernel.org>.
- [21] L. Palopoli, R. L. Cigno, A., and Colombo. Control and optimisation of HCCA 802.11e access scheduling. In *Decision and Control, 2007 Proc. of 46th IEEE Conference on*, pages 4427–4432, Dec. 2007.
- [22] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.
- [23] D. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the Third usenix-osdi*. pub-usenix, feb 1999.
- [24] Y. Wang and K. Lin. Enhancing the real-time capability of the Linux kernel. In *IEEE Real Time Computing Systems and Applications*, Hiroshima, Japan, October 1998.

A The LFSp algorithm

```

1 Parameter: T, T_sample, tol, len;
2 Input: sens(h)
3 Output: Q_req(h)
4 global: c_up(h), c_down(h), W(h), Q(h), dir(h), pos(h)
5
6 /* function body */
7 /*first call */
8 if (h == 0) {
9     dir(0) = UP; pos(0) = 0;
10    c_up(0) = T; c_down(0) = 0; W(0) = 0;
11    Q_req(0) = (c_up(0)+c_down(0))/2;
12    return;

```

```

13  };
14  /* estimation of computation time and workload */
15  if ( sensors(h)==0 ) {
16      W(h) = 0;
17      c_down(h) = c_down(h-1);
18      if ( dir(h-1) == DOWN )
19          if ( pos(h-1) < len )
20              pos(h) = pos(h-1) + 1;
21          else
22              /* all the way down and sensor still 0
23               means c_min overestimated
24               c_down(h) = Q(h-1)/2;
25          else
26              pos(h) = 0;
27              c_up(h) = min(c_up(h-1), Q(h-1));
28              if (c_up(h)-c_down(h) < tol)
29                  c_up(h) = c_down(h) + tol;
30      } else {
31          /* sensor = 1 */
32          if(dir(h-1) == DOWN)
33              pos(h) = 0;
34          dir(h) = UP;
35
36          W(h) = W(h-1) + c_max(h-1)-Q(h-1);
37          if ( sensor(h-1) == 0 ) {
38              c_down(h) = max(c_down(h-1), Q(h-1));
39              if (c_up(h-1) - c_down(h) < tol)
40                  c_down(h) = c_up(h-1) -tol;
41          };
42          if ( pos(h-1) < len )
43              pos(h) = pos(h-1) + 1;
44          else if (W(h) == 0)
45              c_up(h) = (c_up(h) + T) / 2;
46      }
47      /* Computation request */
48      mid_point = ( c_up(h) + c_down(h) )/2;
49      step = (c_up(h) - mid_point)/len;
50      if (dir == UP)
51          end_point = c_up(h)
52      else {
53          end_point = c_down(h);
54          step = -step;
55      };
56      /* estimated computation time */
57      c_est = mid_point + step * end_point;
58      Q_req(h) = c_est + W(h);
59      return;

```