

Adaptive real-time scheduling for legacy applications

Luca Abeni, Luigi Palopoli
University of Trento
luca.abeni@unitn.it, palopoli@disi.unitn.it

Abstract

A remarkable research activity has been carried out in the past few years to support real-time applications by means of appropriate scheduling solutions. For scarcely known or highly dynamic applications, feedback scheduling has emerged as an effective technique to tune the scheduling parameters, based on a run-time monitoring of the timing performance of the application. This technique requires a specialised API for the application and is therefore unfit for legacy applications, for which the source code is not accessible. In this paper, we present an alternative technique, called Legacy Feedback Scheduling (LFS), for feedback based adaptation of the scheduling parameters. LFS does not make any assumption on the programming model of the application and is applicable to legacy applications. Throughout the paper, we carry out a comparison between a well settled technique (called adaptive reservations), which leverages a particular structure for the application, and LFS. The conclusion is that the greater generality and flexibility of the LFS has to be paid in terms of timing performance. However, LFS successfully identifies the “average” scheduling parameters needed to sustain a given execution rate.

1 Introduction

In the 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 inroad 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 [20], to reduce its footprint [15], and to decrease the latency¹ experienced by real-time applications [18].

The evident advantage of adapting a GPOS to embed-

ded and real-time applications is code reuse: a plethora of software components and applications developed for the GPOS, can be used on the embedded devices, with drastic reductions on the development costs and on the time-to-market.

Moving along this avenue, though, developers and system integrators are confronted with a non trivial problem: traditional “legacy” applications are rarely developed with real-time constraints in mind. In fact, the problem of ensuring an acceptable real-time behaviour to a general purpose application is often swept under the carpet relying on the power of the hardware. This brute force solution is simply not affordable in many products for which the cost of the hardware is a significant share of the target price.

On the other hand, decades of research has provided us with effective strategies for real-time computing. Starting from the seminal work of Liu and Layland [12], the use of static or dynamic priorities for real-time schedulers has emerged as a successful paradigm for hard real-time applications. The use of this family of algorithms can be complemented by the application of the real-time theory [7, 13], which provides the mathematical foundations to guarantee that a set of applications can execute respecting their real-time constraints. This approach assumes the knowledge of the worst case workload generated by the application (in terms of Worst Case Execution Time and minimum inter-arrival time between two subsequent activations). Therefore, it is perfectly applicable in the context of embedded controllers, for which applications are perfectly known and execute following “fixed” patterns’.

In more dynamic environments (open systems), we can have applications for which the strict compliance with every deadline is not desirable if the price to pay is a dramatic underutilisation of the system. This consideration applies to multimedia applications (for which a moderate occurrence of deadline misses is not a big problem as long as the application behaves acceptably in the average) but, surprisingly enough, also to some control applications [21, 11]. For this class of applications advanced scheduling solutions have been developed [16, 17, 9, 6, 23, 1]. The design approaches based on these scheduling paradigms [1, 10] enable one to offer differentiated deterministic or stochastic timing guarantees for each application. However, **an a-priori knowledge of the application requirements (e.g., the proba-**

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 real-time theory.

bility distributions of the computation time) is still required, which makes this hardly a viable solution for open systems (in which the activation of an application can be decided dynamically).

In the recent years, this difficulty has been surmounted by applying feedback control to real-time scheduling [2, 3, 14]. The idea is that while the applications execute, their real-time behaviour is monitored and corrective actions are taken changing the scheduling parameters so that specified Quality of Service (QoS) objectives are met. This approach has several advantages. First, it allows the system to identify the execution requirements of the application, and hence its scheduling parameters, even if the application has never been executed before. In other words, the system has self-tuning abilities. Secondly, it allows the system to change the resource allocation in time to accommodate the time-varying execution requirements of data dependent applications.

The main drawback of most of the real-time approaches cited above is that the system has to be aware of the deadlines of the application (or anyway of its timely progress). This is possible only if the application adheres to a well specified real-time tasking model based on a sequence of jobs (each one activated at some time and associated with a deadline). A specialised API, such as those offered in some real-time extensions of GPOSs [24, 26, 19, 8], allows the system to be notified of a job activation and/or termination, thus measuring the scheduling delays and increasing or decreasing the quantity of resources allotted to the task. In [25] the adaptation is made using different parameters (i.e., the real-rate) but the approach is still invasive on the application.

In this paper, we go a step further. In particular, we consider time-sensitive legacy applications. For these class of applications, timing constraints are implicitly present *but* are not exposed to the system by using a specialised API. The challenge we face is to **use a feedback mechanism to adjust the scheduling parameters of legacy applications so that their temporal constraints are respected**. The information used by the system is coarse grained with respect to the algorithms described above. Therefore, the “punctual” performance (intended as the ability to respect every timing constraint) is necessarily lower. However, the technique is totally uninvasive on the applications and it keeps in check their “macroscopic” behaviour reducing the waste of resources (which is often a sufficient result).

The rest of the paper is organised as follows. In Section 2, we show 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 a technique, known as adaptive reservations, that leverages the real-time task model to change the scheduling parameters. In Section 4, we show how a different technique can be applied to legacy applications and, in Section 5, we show experiments obtained with an implementation of both techniques comparing their performance. Finally, in

```

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.

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 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 to easily assign 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

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

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 of “missing” a deadline $Pr\{f_{i,j} \leq d_{i,j}\}$. In this case, the execution of the primitive shown above allows the system to take 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 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 [4], 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, setting aside the classical real-time task model.

3 Adaptive Reservations

Feedback scheduling is a dynamic adaptation of the scheduling parameters (the *controlled values*, or actuators) based on some run-time QoS measurements (the *observed values*) so that some QoS metric is maximised. The actuators are the scheduling parameters decided for each job and the observed values are acquired through appropriate “sensors” inside the Operating System. Adaptive reservations [2, 3] are a simple example of feedback scheduling based on *Resource Reservations*.

A Resource Reservation [17, 23] (Q_i, T_i^s) is an abstraction which can be used to schedule task τ_i , requiring that τ_i is reserved a time Q_i in each period T_i^s . A particular implementation of this paradigm that we will use in this paper is the Constant Bandwidth Server (CBS) algorithm [1], in which a CPU reservation algorithm is imple-

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

mented using dynamic real-time priorities (in particular, the CBS uses EDF priority assignment).

The basic CBS idea is to schedule tasks based on their *scheduling deadlines* d_i^s , with d_i^s increased by T_i^s every time that τ_i executes for a time Q_i (see the original paper for a detailed description of the algorithm - Appendix A contains a short summary). So, the difference between the scheduling deadline at the end of a job and the job’s deadline can be used to measure if a task is reserved enough time: based on this idea, the *scheduling error* $\epsilon_{i,j} = d_i^s - d_{i,j}$ has been defined as an observed value and used to adapt the amount of time Q_i reserved to τ_i . Note that when using adaptive reservations the reservation mechanism is not used to provide temporal isolation (as in the original reservations’ papers), but to control the amount of time assigned to each task (providing an actuator for the feedback mechanism).

Adaptive reservations have been implemented by modifying the `rt_job_finish()` function defined in Figure 1 so that it executes the following steps:

1. Read the scheduling deadline d^s and computes the scheduling error ϵ
2. Compute a new reserved time Q by using a *feedback function* $f(\cdot)$: $Q = f(Q, \epsilon)$
3. If the system is overloaded (that is, $\sum_i \frac{Q_i}{T_i^s} > 1$), apply a *compression function* $g(\cdot)$ so that $\sum_i \frac{Q_i}{T_i^s} = 1$
4. Update the scheduler parameters according to the computed values

A mathematical model of adaptive reservations for a periodic task has been developed in [5]:

$$\epsilon_{i,j+1} = \begin{cases} \epsilon_{i,j} + \frac{c_{i,j+1}}{Q_{i,j+1}} - T_i^s & \epsilon_{i,j} \geq P_i \\ \frac{c_{i,j+1}}{Q_{i,j+1}} - T_i^s & \epsilon_{i,j} < P_i \end{cases}$$

where $\epsilon_{i,j}$ is the scheduling error for task τ_i measured at the end of job $J_{i,j}$. Based on this model, control theory can be used to design a proper feedback function $f(\cdot)$ so that some properties of the closed loop system can be guaranteed [3, 22]. Such CPU controllers have been proved to work very well, and to provide good performance even when the tasks’ parameters are not known a priori.

4 Feedback Scheduling in Legacy Applications

The application of the adaptive reservation technique shown in the previous section requires that:

- the application be logically structured using the real-time task model (i.e., as a stream of jobs);
- appropriate calls to a real-time API be used to delimit the duration of a job.

Indeed, the call to `rt_job_finish()` (or an analogous function) at the end of each job read the scheduling error and execute the feedback function

The problem is that many time-sensitive application satisfy neither of these conditions. Indeed, they have not been designed to run on operating systems explicitly providing real-time support and they are not aware of the concept of job.

In principle, if the source code is accessible, it is possible to extract from the application the notion of a job and to hack the code introducing the appropriate calls. For instance, one can modify a MPEG decoder associating a job with the operations needed to decode a frame and insert a calls to `rt_job_wait()` and `rt_job_finish()` as appropriate. The application of this approach often clashes with non trivial problems. First, the source code of the application and/or the tool chain necessary to generate the code for the target platform may be unavailable or expensive. Second, the refactoring of the application can be a difficult and risky operation, which could tamper with the application functionalities and reduce its robustness.

In contrast, the alternative approach presented in this section applies the feedback scheduling to “unaltered” legacy applications. Clearly, in the absence of a notion of “job”, it is not possible to define a deadline and measure the scheduling error as $\epsilon_i = d_i^s - d_i$.

However, the scheduling deadline d_s^i is still used by the scheduler and, at a generic instant t , the quantity

$$\epsilon_i = d_i^s - t$$

can be related to the progress of the application. For instance, consider a legacy MPEG player application. In this case, we have an implicit temporal constraint on the rate (given by $1/fps$, where fps is the video rate, in frames per second). Therefore, we can choose a server period $T_i^s = P_i = 1/fps$ for the reservation. If at some point in time, we have $\epsilon_i > T_i^s$ then we can deduct that the application has not been given enough time to keep up with the desired rate. This is a direct consequence of the scheduling mechanism of the CBS: if the scheduling deadline is postponed (overcoming T_i^s) it means the task has executed for more that its assigned budget Q_i without incurring a blocking condition.

The feedback scheduling algorithm for legacy applications (called *Legacy Feedback Scheduler - LFS* in the rest of the paper) can be organised as follows:

1. The control algorithm is executed for *all tasks* with a fixed periodicity T^{sample} ;
2. Every T^{sample} time units the scheduling error of each time sensitive task τ_i is sampled;
3. The reserved times $Q = (Q_1, \dots, Q_n)$ are updated as $Q = f(Q, \epsilon)$, where $\epsilon = (\epsilon_1, \dots, \epsilon_n)$;
4. The scheduling parameters of the different tasks are updated.

The first evident difference with the adaptive reservations for real-time applications described in the previous section is in the sampling mechanism. Indeed in the adaptive reservation the scheduling error of a single task is sampled upon each job termination (event triggered sampling). Therefore, each task is associated with a dedicated Single Input Single Output (SISO) controller, which has a limited visibility of the system. In fact, it has visibility only of the evolution of its task. Therefore, we need a global mechanism, the compression function $g()$, which takes a corrective action whenever the different requests of the task controllers violate the global schedulability test. Note that this compression function can be used, in case of system overload, to decide which application is more important and should be assigned more resources.

In contrast, the LFS uses a periodic (time triggered sampling), since the job termination events are no longer defined. Furthermore, the LFS controller has a complete visibility of all tasks. Therefore, it can operate as a Multiple Input Multiple Output (MIMO) controller; note that the MIMO controller can be designed to optimise some global utility function, assigning different importances to the various applications (in practice, we can consider a weighted sum of the QoS experienced by the tasks running in the system). However, this paper does not focus on the controller’s design, and a very simple controller will be used as an example.

It is also important to notice that the event triggered nature of adaptive reservations permits to control the scheduling deadline in the instants when it matters (the job finishing times), whereas the time triggered nature of LFS only permits to control the scheduling deadline in periodic instants that are not necessarily related to the timing evolution of the application.

A second important difference is that in adaptive reservations controllers can rely on a finer grained information. Indeed, the scheduling error used for adaptive reservations can be used to decide if a job received too much, too little or the exact amount of resources. In the case of LFS, we are able to distinguish if the task is receiving too little or not. In other words, the sensor is strongly quantised and produces a binary value.

A third difference is that controllers are activated more frequently (at each job termination) in the adaptive reservations than in the LFS. Therefore, the feedback correction allows us to enforce the deadlines of the subsequent jobs, while maintaining the correct allocation of resources. As a consequence, *LFS can only ensure that each application is executing at the correct rate*, whereas *adaptive reservations are able to ensure that deadlines are actually respected*.

Based on these consideration, not only is it easy to foresee a performance gap with respect to adaptive reservations, but also the controller design turns out to be a non trivial task given the string non linearity introduced by the sensors. Since the main goal of this paper is to introduce the idea of LFS, putting the stress on the differences with

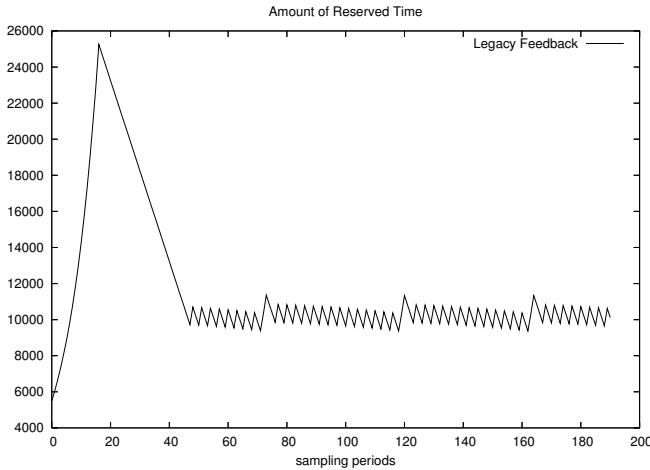


Figure 2. Evolution of the reserved time for a periodic task with $C = 10ms$ and $P = 50ms$.

the adaptive reservations and highlighting the implications on the programming model, we will propose below a very simple algorithm for the feedback function $f()$, reserving for future work the adoption of more sophisticated mechanisms.

Considering the simple case of a single task, whose expected rate is $1/P_i$. If we choose the server period $T_i^s = P_i$, a reasonable behaviour for a feedback function is that it increases Q_i if $\epsilon_i > T_i^s$ (meaning that the task is not keeping up with its required rate), and decrease it otherwise. To avoid excessive periodic fluctuations, we found it convenient to adopt an asymmetric scheme (inspired by the TCP congestion control algorithm) that operates with a multiplicative increase and a subtractive decrease strategy. As a result, the used feedback function is:

$$Q_i = \begin{cases} \alpha Q_i & \epsilon_i > T_i^s \\ Q_i - \beta & \text{otherwise} \end{cases}$$

A low-pass filter has also been added to this feedback function to avoid too many variations in the reserved time. The feedback function can be generalised to the case of multiple tasks using a compression mechanism similar to the one adopted in the adaptive reservations.

5 Experimental Results

To show that the proposed feedback scheme is able to adapt the amount of reserved time for correctly serving a legacy application, some tests on a real implementation (based on a modified version of the Linux kernel) have been performed, using both synthetic load and a real application (a video player). LFS has then been compared with adaptive reservations, to show the performance gap caused by the different type of model adopted in the two cases.

All the experiments have been ran on a PC with an AMD64 CPU at 1.8Ghz and 512MB of RAM, running

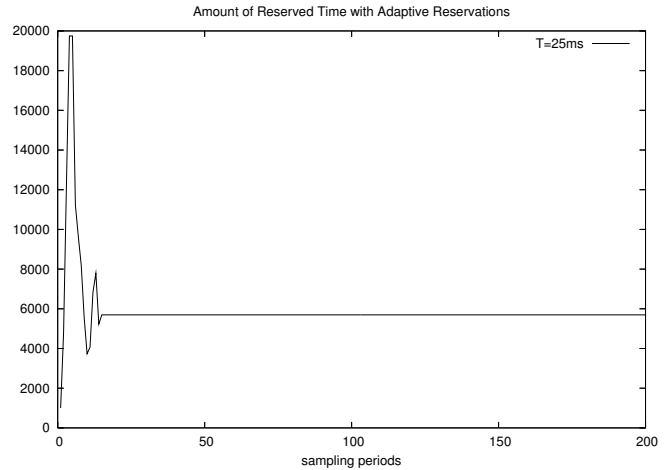


Figure 3. Evolution of the reserved time for a periodic task with $C = 10ms$ and $P = 50ms$ served by an adaptive reservation with $T^s = 25ms$.

a modified version of the 2.6.21.4-rt12-cfs-v17 Linux kernel (the latest RT kernel considered stable by OSADL) with a CBS scheduling module [4] and a middleware implementing the feedback policy.

5.1 Synthetic workload

In the first test, a periodic task with fixed execution time $C = 10ms$ and period $P = 50$ has been served by a CBS with server period $T^s = 50ms$ and reserved time Q controlled by LFS. The evolution of the reserved time is shown in Figure 2: as it is possible to see, Q starts from a small value and continues to increase until the 20th sampling period, then it decreases until it reaches the “correct” time of $10ms$, after the 40th sampling period. After this initial “adaptation phase”, the reserved time continues to oscillate around $Q = 10ms$. These fluctuations are due to the fact that the observed value is coarsely quantised. Therefore, we are able to detect that the amount of reserved CPU is not sufficient only if the scheduling error increases beyond a threshold. This behaviour is similar to the one exhibited by the TCP congestion control algorithm, which causes oscillations in the sending rate because a congestion is only detected after packets are lost (so, the observed value is a again a boolean saying if the connection is congested or not).

LFS has been compared with adaptive reservations; for this purpose, the periodic task used in the first experiment has been instrumented to call `rt_job_finish()` at the end of each periodic execution (which, in this case, clearly qualifies as job), and an adaptive CPU reservation has been used to serve it. Since the previous LFS experiments have been run by using a very simple feedback function, it has been decided to use a very simple feedback function (a PI controller) for adaptive reservations too. The application of PI controllers in adaptive

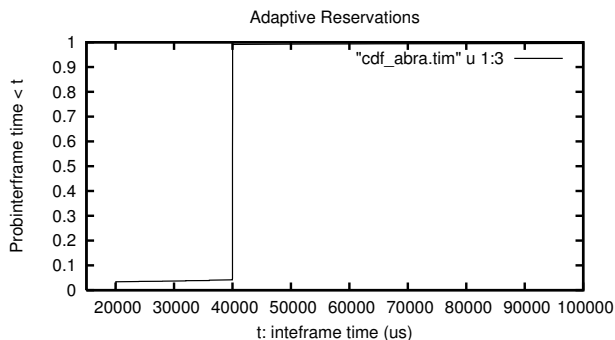


Figure 4. CDF for the inter-frame times with adaptive reservations.

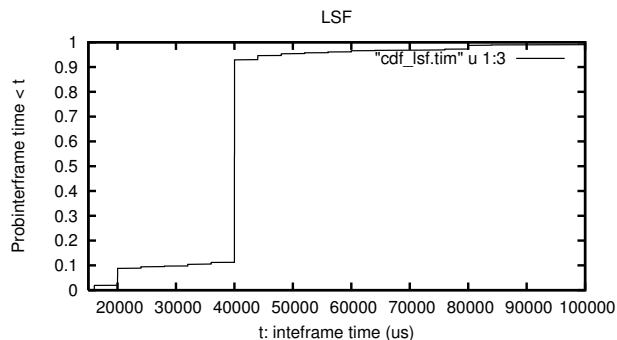


Figure 5. CDF for the inter-frame times with LFS.

reservations has been previously studied [5], showing that the server period T^s must be assigned so that $P = kT^s$, where P is the task period or relative deadline, and k is an integer larger than 1. The analysis also showed that an adaptive reservation will over-estimate the task's execution time, and the amount of over-estimation can be reduced by increasing k . To avoid penalising LFS, the worst possible server period ($k = 2 \Rightarrow T^s = 25000ms$) has been selected for the adaptive reservation.

The evolution of the amount of reserved time Q is plot in Figure 3: after an initial transient in which the reserved time oscillates, around the 15th job Q stabilises on about 5.7ms. This means that $Q = 5.7ms * 2ms = 11.4ms$ are reserved to the task every period and the execution time $C = 10ms$ of a job is overestimated by $5.7 * 2 - 10 = 1.4ms$.

Comparing Figures 2 and 3 it is possible to see that adaptive reservations are able to adapt in a shorter time respect to LFS. The price to be paid is a higher a higher peak of CPU allocation (overshoot): note that the maximum fraction of CPU requested by the adaptive reservation is about $19.7ms/25ms * 100 = 0.78\%$, while the maximum fraction of CPU requested by LFS is about $25.2ms/50ms * 100 = 50.4\%$. More importantly, the adaptive reservation scheme does not exhibit the oscillations presented by LFS. To verify the effect of T^s on the result, the experiment has been repeated with $T^s = 12500ms$ ($k = 4$), verifying that after few server period the reserved time Q is stable around 2.6ms, meaning that a total time of 10.4ms is reserved in each period $P = 50ms$, with a total overestimation of 0.4ms, which is acceptable⁴.

5.2 Video Player

In a second set of experiments, LFS has been tested with a real application: a video player (based on GTK and on the FFmpeg codecs) reading compressed frames from disk, decoding them, and displaying them at 25fps

⁴Reducing the execution time overestimation below 0.4ms is not easy, because of the timer's resolution in Linux.

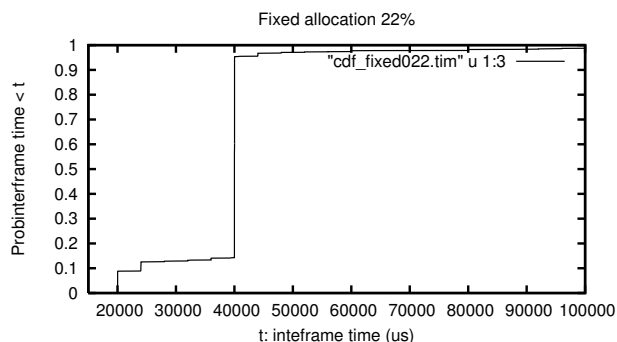


Figure 6. CDF for the inter-frame times with fixed allocation (fine tuned).

(hence, with a period $P = 40ms$). The frame decoding times are obviously variable, and the `top` command indicates that the player consumes about 20% of the CPU time. In this case we considered four different scenarios: A) application of the adaptive reservation on a modified version of the player, B) application of the LFS on the unmodified version of the player, C) static allocation, carefully hand-tuned on the trace to get a good performance, D) static allocation with CPU time roughly equal to the average computation time.

In this case, the quantity of interest was the inter-frame time (i.e., how much time elapses between two frames) and the correct value is clearly 40ms. The ideal configuration for the Cumulative Distribution Function (CDF) for the inter-frame times is a step function at the value $t = 40ms$, i.e., we should have

$$Probability \{Interframe time < t\} = \begin{cases} 0 & t < 40ms \\ 1 & t \geq 40ms \end{cases}$$

In the adaptive reservation case (Figure 4), the performance of the system is close to the ideal expectations. Indeed, we have a first smaller step at 20ms (due to the occasional over-allocation of CPU during the transient phase) but for the greatest part the samples have inter-arrival time equal to 40ms. The application of LFS clearly produces a

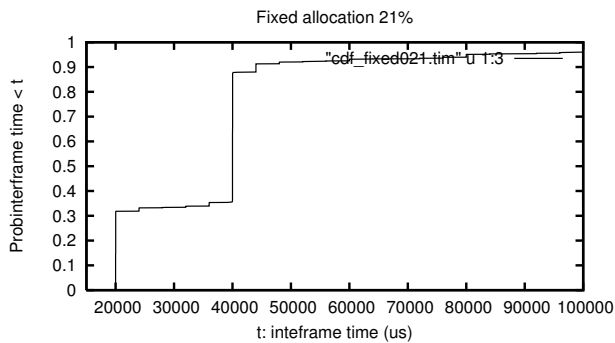


Figure 7. CDF for the inter-frame times with fixed allocation (average).

worse performance (Figure 5). Indeed, we can see a long tail before reaching the value 1 for the probability, meaning that for many samples the inter-frame time is much higher than $40ms$. However, the performance is very similar to the one that we get for a hand tuned static allocation (Figure 6), and turns out to be better performing than a rough static allocation (Figure 7). Indeed, in the latter case, there are samples for which the interframe time is far beyond $100ms$.

The conclusion suggested by the reported experiments are the following. First, the adaptive reservation scheme outperforms any other choice (especially if we consider that we made a “careless” choice either for the control algorithm and for the parameters). Second, the LFS scheme produces the same results as a carefully hand crafted static allocation. This is indicative of a good ability to tune the allocation required to sustain a specified rate. The comparison with the static allocation scenarios cannot be pushed too far, since the best performance that we got required a careful a posteriori analysis of the trace (which is clearly unavailable for the adaptive schemes). Moreover, the performance achieved by a static allocation are not robust. Indeed, even a small variation for the allocated bandwidth (moving from 22% to 21 %) determines evident performance losses.

6 Conclusions and Future Work

This paper introduced a new adaptive scheduling mechanism for legacy applications (the Legacy Feedback Scheduler LFS), which allows a system to self-tune the scheduling parameters for time-sensitive applications that are not developed according to the real-time task model.

Since LFS is not provided with enough information about tasks’ temporal behaviours, its performance is necessarily lower than the one achieved with the adaptive reservations mechanism (that make heavier assumptions on the structure of the tasks).

The performance of LFS and adaptive reservations have been compared on real experiments, showing the ad-

vantages of using a real-time task model, but also that LFS offers a reasonable performance in identifying the CPU allocation needed to control the execution rate of an application.

As a future work, a mathematical model of LFS will be developed, to provide a theoretical foundation for this feedback mechanism. This model will allow us to develop more sophisticated nonlinear control algorithms (e.g., inspired by sliding mode control).

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, T. Cucinotta, G. Lipari, L. Marzario, and L. Palopoli. Qos management through adaptive reservations. *Real-Time Systems Journal*, 29(2-3), March 2005.
- [4] L. Abeni and G. Lipari. Implementing resource reservations in linux. In *Proceedings of Fourth Real-Time Linux Workshop*, Boston, MA, December 2002.
- [5] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole. Analysis of a reservation-based feedback scheduler. In *Proc. of the Real-Time Systems Symposium*, Austin, Texas, November 2002.
- [6] S. Baruah, N. Cohen, C. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 6, 1996.
- [7] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Boston, 1997.
- [8] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. Litmus-rt: A testbed for empirically comparing real-time multiprocessor schedulers. In *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 111–126, Washington, DC, USA, 2006. IEEE Computer Society.
- [9] P. Goyal, X. Guo, and H. M. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *Proceedings of the 2nd OSDI Symposium*, October 1996.
- [10] D.-I. Kang, R. Gerber, and M. Saksena. Parametric design synthesis of distributed embedded systems. *IEEE Trans. Computers*, 49(11):1155–1169, 2000.
- [11] 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.
- [12] C. L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1), 1973.
- [13] J. W. S. Liu. *Real-time systems*. Prentice Hall, 2000.
- [14] 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.
- [15] M. Mackall. Linux-tiny and directions for small systems. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2004.

- [16] 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.
- [17] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves for multimedia operating systems. Technical Report CMU-CS-93-157, Carnegie Mellon University, Pittsburg, May 1993.
- [18] I. Molnar et al. Real-time linux wiki. <http://rt.wiki.kernel.org>.
- [19] S. Oikawa and R. Rajkumar. Linux/RK: A portable resource kernel in Linux. In *Proceedings of the IEEE Real-Time Systems Symposium Work-In-Progress*, Madrid, December 1998.
- [20] Open source automation development lab. <http://www.osadl.org>.
- [21] L. Palopoli, L. Abeni, G. Buttazzo, F. Conticelli, and M. Di Natale. Real-time control system analysis: an integrated approach. *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*, pages 131–140, 2000.
- [22] L. Palopoli, L. Abeni, and G. Lipari. On the application of hybrid control to cpu reservations. In *Hybrid systems Computation and Control (HSCC03)*, Prague, april 2003.
- [23] 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.
- [24] R. R. Rajkumar, L. Abeni, D. de Niz, S. Ghosh, A. Miyoshi, and S. Saewong. Recent developments with linux/rk. In *Proceedings of the Second Real-Time Linux Workshop*, Orlando, Florida, november 2000.
- [25] 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.
- [26] 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.

- 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**)

The algorithm can handle an arbitrary number of tasks, given that $\sum_i U_i \leq 1$.

A The CBS Algorithm

As already explained in Section 3, the CBS is a reservation-based scheduling algorithm characterised by two scheduling parameters (Q_i, T_i^s) (with $U_i = Q_i/T_i^s$) that works by maintaining two variables for every task: the server remaining budget q_i (used for accounting) and the current scheduling deadline d_i^s (used for assigning a priority to the scheduled task and for enforcement). Such variables are updated as follows:

- When τ_i is created, q_i and d_i^s are initialised to 0;
- When τ_i activates at time t , the scheduler checks if the current scheduling deadline can be used (if $q_i < (d_i^s - t)U_i$), otherwise a new scheduling deadline $d_i^s = t + T_i^s$ is generated and q_i is recharged to Q_i ;
- While task τ_i executes, the server budget q_i is decreased as $dq_i = -dt$ (**accounting rule**);