

Adaptive real-time scheduling for legacy applications

Tommaso Cucinotta, Fabio Checconi, Luca Abeni, Luigi Palopoli

Many applications executed nowadays on Personal Computers are multimedia applications characterised by soft real-time constraints. Even if they could benefit from a specialised OS support for real-time computing, the absence of standardised solutions often hinders this possibility. We propose a mechanism to export the benefits of real-time scheduling to *legacy* applications, based on the combination of two techniques: 1) a real-time monitor that observes the sequence of events generated by the application to infer its activation period, 2) a feedback mechanism that adapts the scheduling parameters to ensure a timely execution of the application.

Categories and Subject Descriptors: D4.1 [Operating Systems]: Process Management—*Scheduling*

General Terms: Experimentation, Performance, Measurement

Additional Key Words and Phrases:

1. INTRODUCTION

In recent times, computers have emerged as one of the most effective means to produce, store and distribute multimedia contents. They are increasingly used for video and audio streaming, surveillance, video conferencing. Such applications are usually referred to as *time-sensitive*, meaning that the Quality of Service (QoS) they offer is related to their ability to execute respecting some temporal constraints.

In a modern computing environment, several tasks can be executed concurrently on the same processor (or processors). Therefore a prominent role for the provision of temporal guarantees is taken by the scheduling policy. Common scheduling solutions adopted in General Purpose Operating Systems (GPOSs) do not offer an acceptable performance. Neither are hard real-time scheduling algorithms [Liu and Layland 1973] commonly regarded as appropriate solutions. Indeed, these algorithms guarantee every single deadline by making conservative assumptions on the system workload but do not provide any kind of guarantee in overload conditions. Hard real-time guarantees are not required by multimedia applications: moderate violations of temporal constraints can be easily traded for a more efficient utilisation of the system as far as they are kept under control. For these applications, a superior scheduling choice is offered by such soft real-time schedulers as the *resource reservations* [Rajkumar et al. 1998], which guarantee a share of the CPU time to

Adaptive real-time scheduling for legacy applications

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 20YY ACM 0000-0000/20YY/0000-0001 \$5.00

each task or group of tasks under any workload condition. Even using this solution, though, the selection of appropriate scheduling parameters can be very challenging if the execution requirements of the application are time-varying or unknown. Tentative choices, in this case, can determine either a wasteful utilisation of the system resources or severe degradations of the QoS. This problem is difficult to surmount unless an adaptive scheduling mechanism is used that estimates the application requirements and adapts the scheduling parameters accordingly. Proposals of this kind have been made in the recent past [Abeni and Buttazzo 1999; Abeni et al. 2005] but they make some important and somewhat restrictive assumptions. In particular, applications have to be structured as a sequence of real-time jobs and have to notify the start and the termination of each job to the scheduler by means of an appropriate Application Programming Interfaces (API). This way it is possible to monitor the application and take corrective actions (by adjusting the fraction of resources allocated to the tasks) when a deviation is detected from the desired temporal evolution.

An important practical problem hindering the application of this paradigm is that the real-time support present in modern GPOSs is mostly limited to the POSIX real-time extensions [IEEE 2004]. This API provides such features as fixed priority scheduling, timers and mechanisms for tackling the *priority inversion* problem, which are very useful in embedded control systems, but are generally recognised as unfit for multimedia applications. Only recently has a new generation of specific scheduling solutions for this type of applications made inroad into experimental versions of standard GPOSs¹. The adoption of these solutions into mainstream distributions appears as a natural development for a near future.

The use of a specialised API, when available, is relatively easy for applications developed from scratch. When the source code is available, it is possible to review it inserting the appropriate API calls. In other cases, the source code of the application is unavailable or, more simply, the software producers are not willing to take the risk of an expensive and potentially error-prone refactoring. Therefore, it is easy to predict the presence of a large number of legacy real-time applications for many years to come. In this paper, we use the term legacy applications meaning *applications that are implicitly characterised by temporal constraints, but are not developed using a specific API for real-time computing*. In legacy applications, developers contrive to achieve an acceptable timing behaviour by a large range of heuristic solutions (including a generous use of buffering). The robustness of these solutions can be very low when the system is heavily loaded. Moreover, the presence of large buffers increases the latency of the application and reduces its interactivity.

We make the point that, even for legacy applications, a controllable and robust timing performance is best achieved operating at the *scheduling level*. The purpose of the research presented in this paper is then the development of a scheduling mechanism, called Legacy Feedback scheduler (LFS++), that: 1) extends the benefits of real-time scheduling to legacy applications, and 2) operates in a completely transparent way without requiring any modification to the application. This is a complex and multifaceted problem whose solution requires:

¹An example of this kind is offered by the AQuoSA project (website <http://aquosa.sourceforge.net>) or by the LITMUS-RT project (website www.cs.unc.edu/~anderson/litmus-rt/).

- (1) to correctly infer the activation pattern; multimedia applications are typically periodic, the problem is essentially the inference of the period;
- (2) to estimate (as tightly as possible) the computation requirements.

To identify the period we treat an application as a black box and keep track of a set of events generated by the kernel. The subsequent use of Fourier analysis on the time series of these events allows us to identify the dominant frequencies (peaks in the spectrum), and hence the period of the application. The computing requirements are indirectly identified by accounting for the computation time used by the task in every sampling interval. The estimation of the computation requirements and of the period allows us to identify the fraction of CPU required by the application and make the appropriate choice of scheduling parameters.

As well as providing the algorithmic foundations for this approach, in the paper we show how it can effectively be implemented in the context of the Linux kernel (fitted out with a resource reservation scheduler). We report an extensive experimental validation showing the radical improvement of our solution over the standard scheduler and over previously developed solutions. A very important issue is how to ensure a positive interaction with other mechanisms present in the kernel that can have an impact on the computation time of the applications. As an important example, we consider the power management system, which operates on the frequency of the CPU (thus increasing or decreasing the computation time of the tasks) and plays a fundamental role in mobile devices. In the paper, we show an architectural solution to achieve a graceful integration between our adaptive scheduler and the power manager and offer full experimental evidence of the effectiveness of this combination.

The paper is organised as follows. In Section 2, the related work in the research literature is briefly surveyed. In Section 3, we introduce the terminology on real-time scheduling used throughout the paper and the scheduling solution that we use as a basis for our approach. In Section 4, we describe the problem and provide an overview on our solution. In Section 5, we describe the algorithmic foundations of our approach, while in Section 6 we discuss the most important issues related to the implementation and present our implementation. In Section 7 we offer full experimental evidence of the effectiveness of the approach. Finally, conclusions are drawn in Section 8.

2. RELATED WORK

In the last years there has been a considerable amount of research on how to associate temporal constraints to applications, and to guarantee that such constraints are respected. For example, some solutions derived from real-time theory, such as reservation-based schedulers [Mercer et al. 1993; Rajkumar et al. 1998; Abeni and Buttazzo 1998] have been proposed. Such algorithms enable a fine-grained control on the fraction of CPU time (bandwidth) devoted to each application but the point remains open of how to properly choose the scheduling parameters if the computation requirements are not known and/or change over time.

A popular solution to this problem is the use of some adaptation mechanism. A first possibility of this kind is to perform application-level adaptation [Wüst et al. 2005]. The idea is that in response to the (possibly fluctuating) availability of

resources, the application changes its mode to re-scale the workload it generates. In this paper, we take the complementary approach: resource allocation is adaptively tuned to fit the application requirements (*application-level adaptation*).

The problem of dynamically adapting the amount of CPU time reserved to an application can be addressed by applying feedback control to real-time scheduling, as shown by several authors [Li and Nahrstedt 1998; Abeni et al. 2002; C. Lu and Son 2002; Goel et al. 2004; Abeni et al. 2005]. In such approaches, while the applications execute, their real-time behaviour is monitored and corrective actions are taken by changing the scheduling parameters to meet some QoS related objectives.

Computing models that represent an alternative to the real-time task model have been proposed by different authors. An interesting example is offered by the Timely Computing Base (TCB) model proposed by Verissimo et al. [Verissimo and Casimiro 2002]. The authors proposed also an interesting combination of the TCB with an application-level QoS adaptation [Casimiro and Verissimo 2001] mechanism. However, all of the approaches mentioned above mandate the use of some kind of specialised API within the application, and it is not easy to apply them to applications which have not been explicitly developed to use such APIs. The use of a specialised API is assumed by several authors proposing an Operating System support for multimedia and time-sensitive applications [Leslie et al. 1996; Jones et al. 1996; Krasic et al. 2009].

A piece of work that bears some resemblance with this paper is the one proposed by Steere et al. [Steere et al. 1999], who propose a reservation scheme (based on fixed priorities) implemented in the Linux kernel, and a feedback-based controller to automatically set the scheduling parameters. The authors point out the need for detecting the period, but they do not propose any solution other than the choice of default values. More importantly, their work is based on so called “symbiotic” interfaces, a sort of API used by applications in order to allow external components to monitor their progress. A similar approach is proposed by Eide et al. [Eide et al. 2004], in the context of the QuO framework [Krishnamurthy et al. 2001]. Although the authors claim a “non-invasive” introduction of the adaptation logic for the applications, their approach is clearly targeted at applications constructed using the RT-Corba middleware (in fact an API), which simplifies the interaction with a resource allocation module. In contrast, in our work, the adaptation mechanism is entirely transparent to the applications.

The problem of providing QoS guarantees for legacy applications has been also explored in the networking community. Tsetsekas et al. [Tsetsekas et al. 2001] propose the use of proxy servers to determine the network requirements of Internet applications. The approach is not applicable to CPU allocation.

To the best of our knowledge, the first work providing system support for unmodified (an possibly uncooperative) applications that do not use any specialised API is Redline [Yang et al. 2008], which is based on a reservation-based scheduler and uses some *lightweight specifications* to associate the scheduling parameters to applications. The work presented in this paper is orthogonal to Redline, proposing an adaptive mechanism for automatically inferring the specifications from the applications at run time (note that the specifications required by Redline are system dependent, and can also depend on the applications’ input – for example, the

reservation period for a video player depends on the video frame rate).

From the scheduling point of view, the first technique developed explicitly to support adaptive scheduling of legacy applications is the so called Legacy Feedback Scheduler (LFS) [Abeni and Palopoli 2009]. In the LFS scheme, the scheduler samples a binary variable that simply says whether the task received enough computation in the last period or not. Although we have taken inspiration from this scheme for the scheduler presented in this paper (not surprisingly called LFS++), we use a finer grain for the feedback information (the “sensor” inside the kernel measures the amount of CPU consumed by the task), and the estimation of the period allows us to come up with a more precise estimate for the required bandwidth. Therefore, the application of LFS++ necessarily results in a better QoS.

One of the issues in our paper is the period detection. This problem corresponds to the problem of *pitch detection*, very much studied in the signal processing theory. A first class of algorithms that have been proposed to determine the pitch of a periodic signal works on the waveform in the time domain. In their simplest form, such algorithms just measure the rate of events like zero-crossings or local peaks/dips of the signal (e.g., the Zero-Crossing Rate algorithm [Kedem 1986]). These methods work only with simple waveforms, that present a clear definition of their basic cycle. To deal with complex waveforms, one possibility is to work with the frequency domain representation of the signal (as we do in this paper). Some methods consider the components of the frequency spectrum, trying to infer the relevant harmonics it presents, and reconstructing its fundamental frequency [Piszczalski and Galler 1979]. The algorithm we developed is based on similar principles, but we were able to exploit some characteristics specific to our problem to drastically reduce the complexity of the detection. A possible alternative could be the use of Cepstral analysis [Bogert et al. 1963], which uses a Fourier transform of the spectrum itself, to determine its frequency components and relate them to the time domain signal, but its overhead would be too high for our application.

3. BACKGROUND INFORMATION

Before coming to the specific issues related to the performance of legacy real-time applications, it is useful to provide some basic terminology on the real-time tasking model and on the scheduling algorithm adopted in this work.

3.1 The Real-Time Task Model

In real-time theory, a system is by and large modelled as a set $\Gamma = \{\tau_i\}$ of real-time tasks. 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). A task τ_i is modelled as a sequence of *jobs* and is described by a pair (C_i, P_i) : C_i is the worst-case execution time for the individual jobs of τ_i , and P_i is the minimum inter-arrival time between two consecutive jobs (or the task period in case of periodic tasks). Every job should terminate before the arrival of the next job, an implicit deadline.

3.2 The CBS Scheduler

The scheduling algorithm that we use in this paper belongs to the family of the so called *resource reservation* schedulers. A resource reservation scheduler allows one to allocate to each task τ_i (or to each set of tasks) a computation budget of

Q_i^s time units in every reservation period of duration T_i^s time units. This way, not only can the execution rate be controlled (the task receives a fraction Q_i^s/T_i^s of the CPU time) but also the granularity of the CPU allocation can be decided for every single task by the reservation period T_i^s .

The particular algorithm used in this work to implement the reservation behaviour is a hard-reservation variation [Palopoli et al. 2009] of the Constant Bandwidth Server (CBS) [Abeni and Buttazzo 1998], which implements CPU reservations building on top of an Earliest Deadline First (EDF) scheduler. 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 τ_i executes for a time Q_i^s . The scheduling deadline is used to decide the CPU assignment according to EDF. The algorithm provably enjoys the properties required to resource reservation schedulers if the following *schedulability condition* is met:

$$\sum_{i=1}^N \frac{Q_i^s}{T_i^s} \leq 1. \quad (1)$$

The interested reader is referred to the cited paper for additional details on the rules of the algorithm and on its properties.

4. PROBLEM DESCRIPTION AND SOLUTION OVERVIEW

When we use a CBS scheduler for a real-time task, the problem arises of how to choose the (Q_i^s, T_i^s) parameters so that real-time constraints are met.

The problem has easy solutions if the timing parameters of the task are known a priori. In particular, if the task is periodic with period P_i and if we know its worst case execution time C_i , then we can simply set $T_i^s = P_i$ and $Q_i^s = C_i$ and the task provably meets all of its deadlines [Abeni and Buttazzo 1998]. Alternatively, if we know the probability distributions of the inter-arrival and execution times, the server parameters T_i^s and Q_i^s can be set so that the task meets its deadlines with a minimum guaranteed probability. If a single server is used to schedule multiple tasks, hierarchical scheduling analysis [Mok et al. 2001] can be used to properly assign the scheduling parameters (as far as the timing requirements of all the tasks scheduled through the server are known).

The problem with legacy applications is that we cannot rely on any such prior knowledge of the scheduling parameters, and a wrong choice of the parameters can lead to a severe performance degradation. This is particularly evident for the choice of the budget Q_i^s . Indeed, even assuming a perfect knowledge of the application period, if we choose too small a value for Q_i^s (compared to the average CPU utilisation of the task), the application is likely to receive a very bad QoS. Likewise, choosing a large value of Q_i^s affects adversely the possibility to admit new applications.

Much less obvious but equally relevant can be the detrimental effects of a bad choice for the reservation period T_i^s . This problem was discussed in our previous work [Cucinotta et al. 2009] using an analysis technique inspired to the supply bound function [Lehoczky et al. 1989]. It is very illustrative to report here the correct values of the budget Q_i^s (and hence of the bandwidth B_i^s) required to schedule a simple periodic task with $C_i = 20\text{ms}$, $T_i = 100\text{ms}$. As it is possible to see in Figure 1, the required bandwidth ranges from the correct value (20%) to very

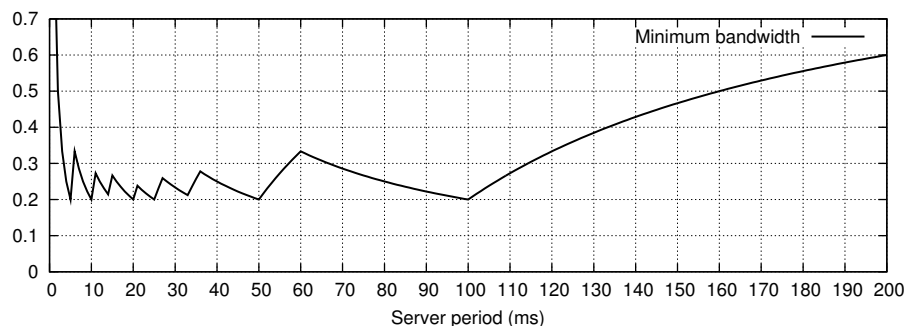


Fig. 1. Fraction of CPU Q_i^s/T_i^s required to correctly schedule a real-time task with 20% utilisation $C = 20$ ms, $P = 100$ ms.

high values (more than 60%) if the server period is chosen too small or too large. The correct bandwidth (20%) is required choosing T_i^s equal to the task period or to a sub-multiple of the task period. However, the choice $T_i^s = P_i$ is the most robust, in that moderate errors in the choice of the period do not lead to an excessive waste of bandwidth. On the contrary if we choose, for instance, $T_i^s = \frac{P_i}{3} = 33\text{ms}$, then even an error of a few milliseconds in the choice of the period easily raises the required bandwidth to a value close to 30% (with an over-allocation of bandwidth close to 50% w.r.t. the task utilisation). These considerations suggest a possible inefficiency in scheduling real-time periodic tasks by a class of algorithms (such as the Proportional Share algorithms), for which the scheduling period is not explicitly considered.

The discussion above leads to the conclusion that an appropriate choice of the scheduling parameters can only be made if we construct a close estimation of P_i (which is usually a fixed parameter) and a statistical estimation of the computation time (e.g., a proper distribution percentile).

4.1 An Example

The theoretical analysis presented in the previous subsection is confirmed by some simple examples with two periodic tasks executed on real hardware.

In different experiments, we have scheduled each of the applications by a reservation with different parameters. The server period T_i^s was chosen arbitrarily *a-priori*, while the budget was dynamically identified by the LFS algorithm [Abeni and Palopoli 2008] to reduce the number of missed deadlines. Figure 2 (a) reports the Cumulative Distribution Function (CDF) of the response-time of one of the periodic tasks (having period $P = 40\text{ms}$), for the different experiments. The figure shows that a server period smaller than or equal to the application period leads to a good performance. Indeed, the CDFs for $T^s < P$ have very short tails after 40ms (a minimum amount of deadline misses is inherent to the way LFS works). However, looking at Figure 2 (b), which reports the corresponding dynamic bandwidth allocations made by LFS, it is clear that the best allocation is the one with the server period equal to the application period, corresponding to a lower bandwidth utilisation of the system.

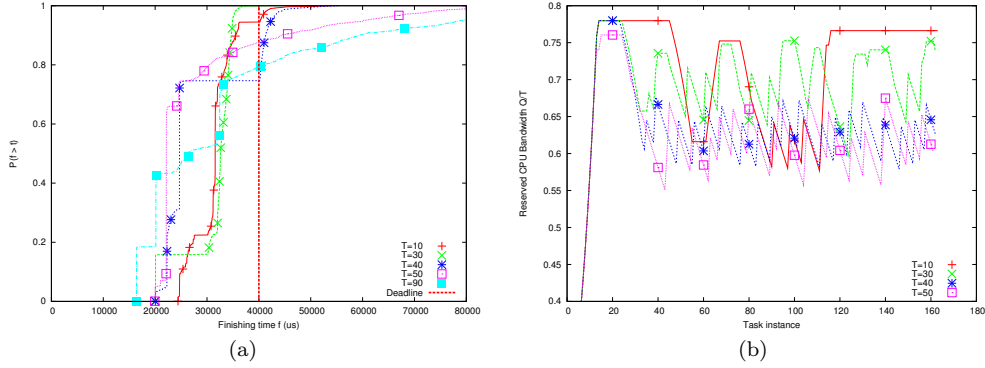


Fig. 2. (a) CDF of the response times for a task with period $P = 40ms$ and various reservation periods T^s . (b) Fraction of bandwidth allocated to a periodic task ($C = 20ms, P = 40ms$) by LFS.

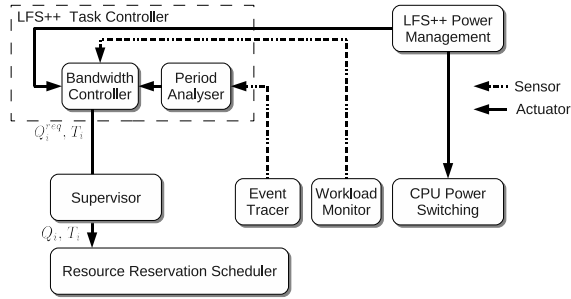


Fig. 3. Block diagram of the algorithmic solution used in LFS++.

The bandwidth waste resulting from the use of integer sub-multiples of the task period as server period, is greater than theoretically foreseen in Section 4. This was also expected, because the discussion in Section 4 refers to the minimum theoretical budget needed to schedule the real-time task hosted by the reservation, while the LFS algorithm approximates (and typically significantly overestimates) this budget.

The experiments above show that the best results, both in terms of application performance, and of allocated bandwidth, are achieved when the server period is chosen in a small neighbourhood of the task period.

4.2 Our Approach

A very high level description of the LFS++ approach is in the block-scheme in Figure 3. The legacy real-time tasks are scheduled through a Resource Reservation scheduler. The correct parameters (Q_i^s, T_i^s) are identified in two steps. In the first one, we use a set of *task controllers*, each one associated with one task, to formulate a request for period T_i^s and the budget Q_i^{req} . Task controllers use only local information on the task. The parameter requests are submitted to the *supervisor*, whose purpose is to enforce the *global* schedulability condition in Equation (1). Namely,

if the requests do not saturate the total available bandwidth, they can be entirely granted $Q_i^s = Q_i^{req}$. Otherwise they have to be curbed to fit in the bound.

The task controller consists of two blocks. The first block (period analyser) constructs an estimation of the task period from a sequence of events traced in the kernel. The second block (bandwidth controller) uses as input the time consumed by each task during the last sampling period. This information is combined with the estimated period to identify a correct pair of reservation parameters. The bandwidth controller keeps track of the past evolution of the task computation time.

For systems with dynamic CPU frequency scaling capabilities, possible changes in the CPU speed (e.g., for power saving purposes), need to be coordinated with such a feedback-based control logics, lest the temporary loss of the provided guarantees. Therefore, we introduced a new block (power management) that interacts with the pre-existing power-management mechanisms present in the kernel.

The design of the supervisor block has been discussed in depth in previous work [Palopoli et al. 2009] and it does not require any additional remark in this context. The design of the other blocks, on the contrary, raises important theoretical and architectural issues tapping several disciplines. In particular, the identification of the task parameters carried out by the task controller is essentially a theoretical problem, which can be solved without any commitment to a specific architectural choice. On the contrary, the design of the event tracing mechanism, of the workload monitor and of the power management block is, by a large extent, architecture dependent.

5. IDENTIFYING THE TASK PARAMETERS

In this section, we dwell on the algorithmic foundation that underpins the design of the task controller. In essence, we assume the presence of some kernel or middleware components that operate as “sensors” providing the period analyser and the bandwidth controller with the necessary input. Although the definition of the algorithms can be made abstracting from the source of the input data, the quality of the result heavily depends on the accuracy and on the frequency with which this information is collected (as clearly displayed by the experimental results at the end of the paper).

5.1 The Period Analyser

The activation of a job for a task is, in a GPOS, associated with a state transition from `BLOCKED` to `READY`. This event is said a *wakeup event* (WKE). In the idealised situation in which a job activation is the only WKE, the period detection would simply amount to identifying the interval between two consecutive occurrences of this event.

In real applications, unfortunately, we can have many of these events even during a job execution (e.g., to perform I/O operations, or to access mutually exclusive memory areas). Generally speaking, by the term *scheduling related event* (SRE) we denote any event that could potentially be associated to a state transition of the task. We make the reasonable assumption that the real-time application generates periodic bursts of SREs, and that the bursts are mostly concentrated at the beginning and at the end of the period to perform the I/O operations. To show

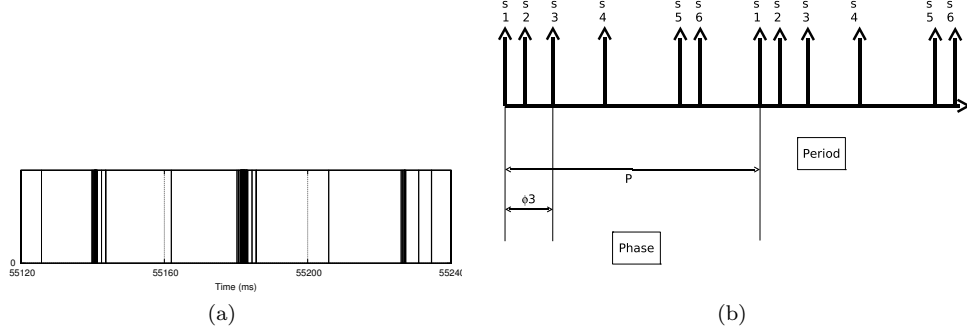


Fig. 4. (a) a sequence of events associated to a segment of execution of an application, (a) The mathematical model as a sequence of Dirac's δ .

that this assumption is well founded, consider the excerpt of a trace recorded for a real application and reported in Figure 4.(a), where each event is represented as a vertical line. As it is possible to see, most of the events are indeed accumulated at the beginning and at the end of the period.

A possible way for modelling this behaviour is to conceptually associate each event at time t_0 with a Dirac delta at time t_0 , denoted by $\delta(t - t_0)$. Therefore, if s_i symbolically represents an event, its periodic repetition can be modelled as a train of Dirac δ : $s_i(t) = \sum_{h=-\infty}^{\infty} \delta(t - \phi_i + hP)$, where ϕ_i is the temporal offset (phase) of the event inside the period (see Figure 4.(b)). A trace can then be modelled as the sum of all signals s_i : $s(t) = \sum_{i=1}^K s_i(t)$, K being the total number of events within each periodic activation of the task. Because of the bursty nature of the events, phases are very close to the start time (0) and to the finishing time (P) of each job.

The Fourier Transform of $s(t)$ turns out to be:

$$S(f) = \mathcal{F}(s(t)) = \frac{1}{P} \sum_{i=1}^K \sum_{n=-\infty}^{\infty} e^{-jn2\pi f_0 \phi_i} \delta(f - nf_0).$$

where f is the frequency variable and $f_0 = 1/P$. Now, suppose that the observation horizon H is limited to L sampling periods ($H = LP$). We can model this effect multiplying the signal $s(t)$ by $G_H(t - \frac{H}{2})$, where:

$$G_H(t) = \begin{cases} 1 & \text{if } |t| \leq H/2 \\ 0 & \text{Otherwise.} \end{cases}$$

Applying standard arguments of signals and systems theory, we get:

$$S(f) = \frac{H}{P} e^{-j2\pi f H/2} \sum_{i=1}^K \sum_{n=-\infty}^{\infty} e^{-jn2\pi f_0 \phi_i} \text{sinc}(\pi(f - nf_0)H). \quad (2)$$

where $\text{sinc}(x) = \sin(x)/x$.

The right hand side of Equation (2) consists of a sum of complex vectors with amplitude $\text{sinc}(\pi(f - nf_0)H)$ and phase given by the direction of the complex number $e^{-jn2\pi f_0 \phi_i}$. Because the values of the phases ϕ are close to 0 or to P , the

vectors are almost “collinear” and the amplitude of the sum is approximatively equal to the sum of the amplitudes. Considering that each sinc function has the highest peak when its argument is equal to 0, the amplitude spectrum of the signal has the peaks in $f = nf_0$. Hence, the distance between $f_0 = 1/P$ between two adjacent peaks can be used to estimate the period of the task. To summarise, the problem of identifying the period of the task amounts to: 1) computing the spectrum of the signal $s(t)$, 2) estimating its peaks and their distance.

5.2 Computation of the Spectrum

The spectrum is computed in the range of frequency $[f_{min}, f_{max}]$ with a step δf . This computation can be made iteratively. Indeed, whenever we record the i^{th} event at time \bar{t}_i , we can model it as a Dirac $\delta(t - \bar{t}_i)$ whose contribution to the spectrum, for each frequency f , is $\mathcal{F}(\delta(t - \bar{t}_i)) = e^{-j2\pi f\bar{t}_i} = \cos(2\pi f\bar{t}_i) - j \sin(2\pi f\bar{t}_i)$. The number of samples to be computed for each of these components of the spectrum is given by $\frac{f_{max} - f_{min}}{\delta f}$. Therefore, the number O of complex exponentiations to perform is:

$$O = \frac{f_{max} - f_{min}}{\delta f} N \equiv \frac{f_{max} - f_{min}}{\delta f} \frac{H}{P} K, \quad (3)$$

where H is the observation time horizon, P is the application period and K is the number of events recorded in each application period.

5.2.1 Peak Detection Heuristic. Instrumental to the determination of the period is a heuristic algorithm to detect the peaks in the computed spectrum. The algorithm is structured as follows:

- (1) compute a sampling of the amplitude spectrum $S(f)$ of the signal $s(t)G_H(t - H/2)$ (the modulus of its Fourier Transform) in the frequency range $[f_{min}, f_{max}]$, with step δf , as discussed above:

$$|S(f)| = \left| \sum_{i=1}^N e^{-j2\pi f\bar{t}_i} \right|; \quad (4)$$

- (2) identify a first set of peaks f_1, \dots, f_m as the local maxima of the amplitude spectrum in the range (ordered by frequency);
- (3) discard all peaks f_i for which $S(f_i)$ is lower than α times its average value \bar{S} (with α configurable);
- (4) if the resulting set of candidate values is empty, then declare the application as non-periodic and **terminate**;
- (5) for each candidate frequency f_i , compute the sum Σ_i of the amplitude spectrum in correspondence of at most k^{max} integer multiples of f_i (set to 10 in the experiments), with a tolerance of ϵ , i.e., compute:

$$\Sigma_i = \sum_{\substack{f_j \in [hf_i - \epsilon, hf_i + \epsilon] \\ h \in \{1, \dots, k^{max}\}, f_j \leq f_{max}}} |S(f_j)|.$$

- (6) select the frequency f_i corresponding to the highest Σ_i value.

The rationale of this algorithm is explained next. In the computation of the spectrum, due to the behaviour of the sinc function and to the inexact adherence of our model with the real signal, we have got a combination of main peaks and of secondary peaks. Our objective is then to identify the main peaks and estimate their distance. More simply, we can identify the first main peak at a frequency greater than 0 and take its value. Indeed, one of the main peaks is necessarily at frequency 0 and therefore the value of the first non zero main peak is itself the distance between two main peaks. The first three steps allows us to identify the candidate peaks and to rule out the evident secondary peaks using an empirical threshold α . If no peak is left, we can conclude that the signal does not possess any periodic structure. Otherwise, we carry out a further analysis step considering that if we identified the first main peak, then further main peaks are expected to be at integer multiples of its frequency. Therefore, we accumulate the spectrum of all these frequencies using a tolerance ϵ (to account for the fact that the peak could not be exactly at the expected frequency) and limiting the number of considered frequencies to k^{max} . This prevents secondary peaks at lower frequencies from outweighing main ones at higher frequencies due to the higher number of accumulated values.

5.2.1.1 *Heuristic Complexity.* The complexity for the frequency detection heuristic is expressed in terms of number of frequencies over the computed transform that need to be scanned. Let $F \triangleq \frac{f_{max}-f_{min}}{\delta f}$ be the number of computed samples for $|S(f)|$. The second and the third steps of the algorithm require the analysis of all the samples. Then (step 5), for each candidate peak frequency f_i , the values of the transform in correspondence of the integer multiples of f_i , with a tolerance of ϵ , are summed up, up to f_{max} . The number of sums to make is given by $\min \left\{ \frac{f_{max}-f_i}{f_i}, k^{max} \right\} \frac{\epsilon}{\delta \omega}$; the final choice of the main peak is immediate and does not have any impact on the complexity. Therefore, the number E of considered elements in the frequency transform is bounded by:

$$E = \frac{f_{max} - f_{min}}{\delta f} + \sum_{f_i \in F_{max}} \min \left\{ \frac{f_{max} - f_i}{f_i}, k^{max} \right\} \frac{\epsilon}{\delta f}, \quad (5)$$

where F_{max} is the set of candidate peaks after step 3.

5.3 The Bandwidth Controller

The LFS++ Bandwidth controller requires the presence of an appropriate “sensor” inside the kernel that measures the CPU time consumed by the application in each interval. This information is fed into a “predictor” or “estimator” component which may easily determine the budget that best suites the application needs, based on the observation of past computation times of the application.

The sensor is sampled periodically and its reading is used to estimate the duration of each job. More precisely, let P denote the application period (estimated by the period analyser), and let S denote the sampling period of the task controller. For the sake of simplicity, assume that S is equal to an integer multiple of P . Let W_k denote the measured time at the k^{th} activation of the feedback loop, W_{k-1} denote the time measured at the previous activation. Then, the new budget Q_k^{req} required

for the next sampling interval is determined as follows:

$$Q_k^{req} = (1 + x) P \frac{\mathcal{P}(W_k - W_{k-1})}{S},$$

where x is called “spread factor” and is set usually between 10% and 20%, and $\mathcal{P}(\cdot)$ is a prediction function returning the computation time expected for the next sampling period. The idea is to translate the expected application workload into the bandwidth allocated by the reservation (the reservation period is set equal to the task period, therefore Q_k^{req}/P is the bandwidth requested by the controller). The predictor \mathcal{P} can be implemented in different ways. In this paper, we propose a “percentile estimator”, which basically memorises the sequence of the past N observed samples, and outputs the estimated p^{th} percentile of the computation times distribution. This may be easily accomplished for p values which correspond to a probability expressed as $\frac{N-j}{N}$, where j is an integer. For example, with $N = 16$, if $p = 1.0$ ($j = 0$) then one has to take the maximum over the last N samples. For $p = 0.9375$ ($j = 1$), one has to take the second maximum, and so forth.

The factor x (which is typically small) increases the bandwidth assigned to the task from the “ideal” assignment (the task utilisation). This factor is needed for two reasons: 1) it enhances the robustness of the control action with respect to prediction errors, 2) it increases the responsiveness of the system to changes in the workload.

6. THE PROPOSED ARCHITECTURE

The implementation of the scheme advocated in this paper requires: 1) the design of an appropriate architecture based on kernel-specific solutions, 2) the availability of the architecture-dependent blocks in Figure 3, namely the resource reservation scheduler, the event tracer, the workload monitor and the power manager. In this section, we briefly describe the architectural implementation, based on the Linux kernel, that we have developed and used for the experiments in the next section.

As far as the scheduling mechanism is concerned, we have used the AQuoSA architecture², which extends the Linux kernel with a Resource Reservation scheduling mechanism. The task controllers and the supervisor are implemented in a user-space daemon, which periodically collects information from the sensing blocks and performs the computation of the algorithms described in the previous section. This daemon interacts with the standard mechanisms present in the Linux kernel to carry out frequency switches on the CPU. In the rest of the section, we will shortly illustrate the design issues for the sensing block and for the power management components.

6.1 Workload Monitor

The workload monitor measures the time a task executes over an interval of time. For POSIX compliant systems (such as Linux) a sensor of this kind is the `clock_gettime()` system call that measures the so called `CLOCK_PROCESS_CPUTIME_ID` and the `CLOCK_THREAD_CPUTIME_ID` clock values, providing us exactly with the information we need at the granularity level of the process

²<http://aquosa.sourceforge.net/>

or of the thread. In our specific case, we used the API of the AQuoSA middleware and in particular the AQuoSA API call `qres.get_time()`, which returns the CPU time consumed by all the threads attached to a CBS since its creation.

6.2 The Event Tracer

The purpose of the event tracer is to pinpoint periodic events generated in the execution of a task. Any mechanism created to this purpose can be evaluated along two different dimensions: its accuracy and its intrusiveness. By accuracy we mean the ability to detect *exactly* events that are repeated with regular temporal patterns, limiting the addition of noisy measurements. By intrusiveness, we mean the extent of the modification on the basic structure of the kernel required to extract the information. We have identified two solutions that strike a different balance between these two conflicting metrics.

6.2.1 Tracing the system calls. The first solution is based on the consideration that periodic events are often associated to system calls. For instance, a typical periodic task executes some operations (the task body) and then switches to a “blocked” scheduling condition as a result of the execution of a blocking system call (such as `clock_nanosleep()`). In a simplified view, if we knew exactly the primitive used by the task to end a job, we could in principle trace its execution instants and extract a sequence of events which, with a good approximation, can be regarded as periodic. In fact, for a legacy application things are more complicated because we do not know which call is used to block the task, and the same call could be used for different purposes. For this reason, we need to trace the application for the use of a wide set of potentially blocking system calls. This tracing mechanism necessarily produces “noisy” measurements: only a very strict subset of the system calls are in fact periodic. On the other hand, this tracing mechanism can be implemented in a flexible way. In our previous works [Cucinotta et al. 2010; Cucinotta et al. 2009], we have shown both a kernel-based implementation, and an entirely user-space one relying on the Linux `ptrace()` system call. Furthermore, as the standard libraries implementing system calls are usually dynamically linked by applications, it is also possible to exploit dynamic linking in order to intercept the calls. A user-space implementation has the advantage of enabling the tracing mechanism without the need for root privileges.

6.2.2 Tracing the Wakeup Events. The tracing of the WKEs leads to more accurate results as compared to the tracing of the system calls and to other SREs. In fact, the activation of a job is invariably associated to a WKE. Therefore, any periodic task certainly generates a periodic stream of events. Of course, there could be other WKEs in the different jobs of a task that are not repeated periodically (e.g., for occasional synchronisations or I/O operations). However, the jitter introduced in the measurement of a WKE is necessarily smaller than the one introduced by recording the exit of the system call associated with the event. Indeed, the system call exit time may only be recorded when the process is actually scheduled by the kernel, while wake-up events are recorded inside the kernel, without any scheduling delay. Also, the number of expected WKEs is considerably smaller than the one of other SREs, with an evident reduction of the overhead associated to the period detection mechanism, which needs to process such events.

On the other hand, the tracing of WKEs is necessarily more intrusive: it has to be carried out forcibly inside the kernel using root privileges. The potential security risk is, in our case, alleviated by the use of a system daemon, which manages all the different users and can implement a variety of security policies (e.g., a maximum bandwidth “quota” allowed to un-privileged users to prevent denial of service attacks). This daemon is the only entity in the architecture that requires to execute with root privileges.

A first possibility for tracing the WKEs in Linux is offered by the `ftrace` kernel-level tracer. This tool was designed for debugging purposes, and is unfit for a “production” installations of the OS. Indeed, it traces much more events than we need (not only wake-ups, but also sleeps, signal reception, and preemption times) and produces a formatted output (wasteful to create and to parse). For these reasons, `ftrace` is hardly an acceptable option in our context.

Therefore, we have extended our `qtrace` Linux kernel patch (initially used for the system calls) to trace processes wake-up events as well. This patch has been developed along the following lines

- a special device allows a user-space program to specify the tasks for which the tracing of WKEs has to be enabled;
- a circular memory buffer is used within the kernel to record wake-up times of the traced tasks;
- a user-space program (in our case the LFS++ daemon) periodically depletes the circular buffer communicating the traces to the period analyser.

As shown in Section 7, the `qtrace` patch allows for substantial overhead savings with respect to the use of the `ftrace` tracer.

6.3 Power Management

Most computing systems nowadays possess power-management capabilities, which are largely supported by GPOSSs. A popular approach to manage power is by operating on the CPU frequency. This can be done using kernel-level components (e.g., the Linux “governors”), or user-space daemons (e.g., `cpuspeed` or `powernowd`), which execute asynchronously with respect to the applications. Unfortunately, by changing the CPU frequency the workload generated by the real-time applications changes and the QoS performance provided by the LFS++ framework is at serious risk of being disrupted. More specifically, increasing the CPU frequency cannot disrupt the task QoS since the CPU utilisation of the task (the ratio between computation time and period) decreases. The transient over-provisioning of bandwidth is, in this case, gradually compensated by the feedback. On the contrary, a frequency decrease corresponds to an increase of the task utilisation. Until the LFS++ performs a bandwidth adjustment, the real-time application accumulates delay and the QoS degradation can be severe. The problem here is that the LFS++ Bandwidth Controller described in Section 5.3 uses observations on the past activations of the task and can react slowly to sudden utilisation changes such as the one introduced by a CPU frequency switch.

To deal with this problem we integrated a power management algorithm into the LFS++ daemon to implement a QoS sensitive power management policy organised

as follows. The daemon periodically monitors the system workload and utilises the `powernowd` power-management algorithm, to keep the overall system utilization in a target configurable interval (defaulting to the [20%, 80%] range). Whenever a CPU frequency increase is required by the algorithm, it is actuated immediately. More complex is the management of a frequency decrease request. In this case, the daemon performs a rough “projection” of the expected utilisation of the controlled real-time tasks with the new target frequency and:

- if the projected utilisation overcomes the schedulability bound in Equation (1), then the frequency switch is dismissed;
- otherwise, a *mode-change protocol* is engaged by the daemon: the budgets assigned to the controlled tasks are first increased according to the projected utilisation and the CPU frequency switch is delayed to the time instant in which all the new budgets have been changed on the underlying scheduler (this amounts to waiting at most for a time duration equal to the maximum reservation period among the ones active in the system).

After a frequency switch, the history of computation times collected by the bandwidth controllers is reset to a single sample equal to the projected budget value, so as to allow the control loops to promptly adapt to the changed workload.

The algorithm for projecting utilisation figures across CPU frequencies is very simple. Given the current frequency f_{curr} at which the current utilisation U_{curr} has been observed, and the target frequency f_{dest} , the projected utilization U_{dest} is roughly estimated as:

$$U_{dest} = U_{curr} \frac{f_{curr}}{f_{dest}} \quad (6)$$

Although this approach may seem a naive one, it is sufficient for our needs since fine-grained adjustments of the budget can be deferred to the subsequent executions of the LFS++ bandwidth controller.

The inclusion of power-awareness within LFS++ introduces substantial improvements in the promptness of the mechanism in reacting to frequency changes. As shown in Section 7, not only does this mechanism radically reduce the transient QoS degradation when the frequency is reduced, but it also ensures a quicker convergence of the bandwidth controller when the frequency increases, enhancing its efficiency.

As a final remark, the choice of `powernowd` for the power adaptation logic is incidental. We believe that a similar solution could be found with other and possibly more sophisticated (and/or effective) algorithms for power management.

7. EXPERIMENTAL RESULTS

The techniques and architecture described in the previous sections have been implemented in a Linux-based system. The implementation, comprising the necessary Linux kernel patches and user-space tools, is available for download at the following URL: <http://retis.sssup.it/people/tommaso/papers/acmtecs10> and has been used to demonstrate the effectiveness of our approach through an extensive set of experiments. First of all, LFS++ is compared with the standard Linux scheduler, showing the advantages when scheduling common multimedia applications under

load. A second batch of experiments shows that LFS++ can reduce the application’s response times even when multiple instances of a real-time application are simultaneously active. The tracing mechanism presented in this paper (which traces the WKEs) is then compared with the one presented in our previous work (which traces the system calls). The fourth set of experiments shows the effectiveness of the LFS++/Power integration (see Section 6.3) on platforms with power-switching capabilities. Finally, overhead measurements are briefly summarised.

7.1 Comparing LFS++ and Linux

In this section, we show the advantages of using LFS++ to schedule multimedia applications over relying on the default general-purpose scheduler provided by the Linux kernel. This is done in the context of a video player application, by measuring the amount of desynchronisation between the reproduced audio and video.

To this purpose, we launched `mplayer`, a video player, showing a segment of the “Big Buck Bunny” movie³, containing H.264 video (encoded at 25fps , with frame size 1920×1080), and AAC audio. The movie was played under different load conditions, and the player was modified so as to output the difference between the presentation timestamps (PTS) of the currently reproduced audio and video frames, referred to as Audio/Video (A/V) desynchronisation. Such value is representative of the annoyance directly perceived by the user during the playback.

As a background workload, we launched `eclipse`, a commonly used development IDE environment based on Java. Experiments have been performed on an Intel(R) Core(TM)2 Duo P9600 CPU running at 2.66 GHz. The IDE environment was taking about 15 seconds to start-up on the considered hardware.

The obtained A/V desynchronisation value during the first 30 seconds of the play are shown in Figure 5. First, we ran the player alone, obtaining a practically flat curve (the one labelled as “No Load”) for the A/V desynchronisation, except for an initial segment due to the transitory period during which the application fine-tunes its internal buffering control loop.

Then, we repeated the play, launching the IDE application at time $t = 2s$, obtaining an increasing desynchronisation (curve labelled as “Load - Linux”) below $0.4s$ between $t = 5s$ and $t = 10s$, followed by a steep increase towards the peak of nearly $1.9s$ of desynchronisation at about $t = 14s$. Finally, when the background application ends the start-up phase, the desynchronisation decreases back to negligible values close to 0.

Then, we repeated the experiment launching the player application under supervision of LFS++. We obtained a desynchronisation (curve labelled as “Load - LFS++”) which was almost entirely below the value of $0.1s$, with the exception of a little peak of $0.2s$ at about $t = 17s$. The achieved maximum peak of the A/V desynchronisation experienced by `mplayer` when under LFS++ was about *one tenth* of the value experienced when using the standard Linux scheduler (e.g., a reduction of the 90% of the A/V desynchronisation).

³<http://www.bigbuckbunny.org>

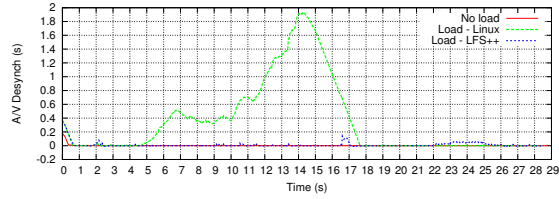


Fig. 5. A/V desynchronisation of `mplayer` under various conditions.

Table I. Parameters of the launched `rt-app` instances.

Application	Period (μs)	Iterations	Load (%)
<code>rt-app</code>	3505	6000	30%
<code>rt-app</code>	8220	13000	28%
<code>rt-app</code>	100000	120000	21%

7.2 LFS++ and Multiple Real-Time Applications

After comparing LFS++ with the Linux scheduler when only a single time-sensitive application is present in the system, we report results showing how LFS++ behaves in presence of multiple periodic real-time applications which need to be run on a system.

To this purpose, we used `rt-app`, a synthetic periodic application we developed that activates periodically at a given period, wasting CPU cycles by running a certain number of iterations of a meaningless loop, then recording the achieved job response time, and going to sleep until the next activation. Job response times are recorded in a static memory array, for being logged onto a file at the end of the program. For such an application, we assumed a relative deadline equal to the activation period.

We ran concurrently 3 instances of `rt-app`, with different periods and number of cycles tuned so as to achieve about 80% of overall system load (roughly measured with `top`). The exact parameters of the launched applications are summarized in Table I. Under heterogeneous periodicity of the task set, the tasks that are most at risk are of course the ones with the shortest periods, thus possessing the most strict timing requirements.

In Figure 6, we report the Cumulative Distribution Function (CDF) of the response-times obtained for the application with the smallest period when launching the three applications at the same time, scheduled by the standard Linux scheduling policies and by LFS++.

When using the default `SCHED_OTHER` Linux scheduling policy (curve labelled as “Linux”), the response times exhibit a quite large variability, with nearly 57% of the values below the deadline of $3.5ms$ (vertical curve labelled as “Deadline”). For most soft real-time applications, a 43% of deadline misses is hardly tolerable, and the application should better be dropped by the system.

Scheduling the same task set with LFS++ achieves a much more stable set of response times (curve labelled as “LFS++”), which remain for nearly the 93% below the $1.5ms$ value, equal to half of the relative deadline (which is equal to

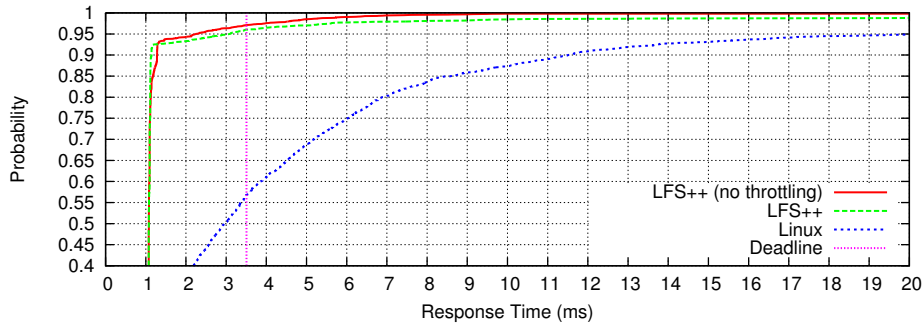


Fig. 6. Response-time CDF obtained under LFS++ compared with the one obtained under the Linux default scheduling policy. .

the period). Still, about 5% of the jobs exhibit a deadline miss, with a relative reduction of the deadline miss ratio of nearly 75% as compared to Linux. However, it is noticeable that the CDF in this case highlights a few residual response time values (below 2%) which are far beyond the deadline. This is due to the interference of the real-time throttling mechanism in the Linux kernel, which by default limits the maximum rough utilization exploitable by tasks at real-time priorities at $950ms$ every $1s$ (this is a security/stability feature which allows for not losing control of the machine in case a real-time task goes into an endless loop due to a bug). In case such a mechanism fires while a job of this `rt-app` instance is running, then such job completion is delayed until the next throttling period, which may be as distant as $50ms$ apart. In fact, disabling the real-time throttling, such phenomenon vanishes (curve labelled as “LFS++ (no throttling)”), with a maximum response time that is about two periods and a half.

7.3 Tracing of Wakeup Events

As mentioned, in this paper we introduced a novel tracing mechanism based on detection of wakeup events at the kernel level, as opposed to the system call events that we used in our prior works [Cucinotta et al. 2009; Cucinotta et al. 2010].

The new tracing mechanism has been tested and compared with system call tracing by using `mplayer` and the “Big Buck Bunny” movie (which is a $25fps$ video, as mentioned in Section 7.1). The two different tracers have been used to infer the player’s periodicity under different system loads and using different observation intervals, and the results are shown in Figures 7 (a) and (b).

In the first figure, we show the average and standard deviation of the detected frequency, at varying observation times. While starting from an observation interval of $0.8s$ both the two tracers allow to infer correct information (consistently with the results presented in previous works [Cucinotta et al. 2010]), for small observation intervals the frequency detected by tracing wakeup events is much more stable than the one detected by tracing system calls.

The improvement due to tracing wakeup events can also be appreciated when considering the overhead dimension. In Figure 7 (b), we plot the number of detected events versus the standard deviation of the detected frequency, for various

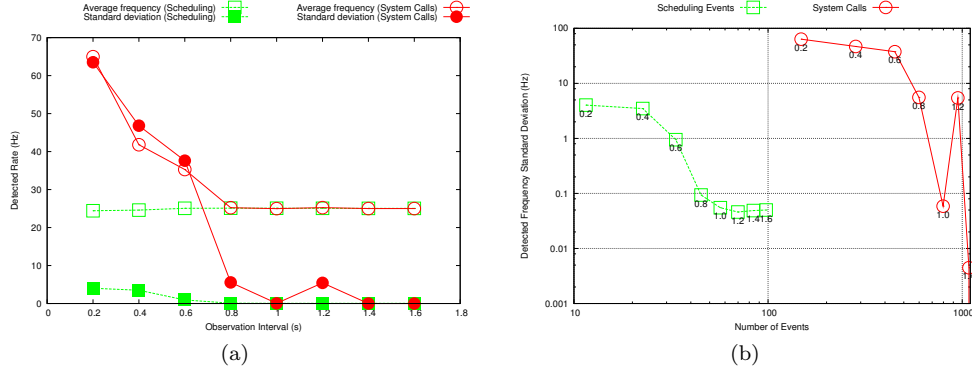


Fig. 7. (a) Detected frequency statistics in the cases of tracing the wakeup events and the system calls, at varying observation times. (b) Number of detected events vs detected frequency precision, at varying observation times.

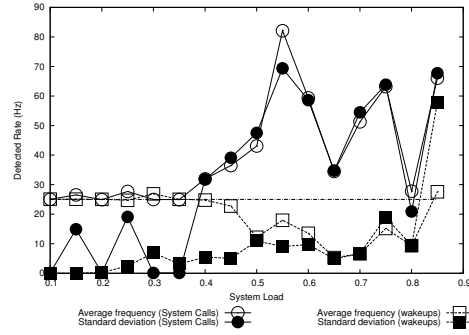


Fig. 8. Detected frequency statistics for different system loads.

observation times. As it can be seen, using the wakeup events leads to a much lower number of events to process and a small standard deviation. When tracing system calls, the standard deviation of the detected frequency can be reduced by increasing the observation interval, but this results in a large number of events to be processed (note the logarithmic scale on both axes).

It is noteworthy to mention that the number of events to process is reduced of barely one order of magnitude (the horizontal axis is logarithmic), leading to a dramatic improvement in the overhead associated to the computation of the frequency transform, which is linearly dependent on such value (see Equation (3)).

Further experiments showed that the presence of some real-time load in the system decreases the precision of the rate detection mechanism, but the mechanism based on wakeup events tracing is affected less than the mechanism based on system calls tracing. For example, Figure 8 shows the detected frequency for an observation interval of $0.6s$ and an increasing system load. It is possible to notice that, when the system load grows beyond the 35% threshold, the event tracing mechanism is not accurate anymore. The mechanism based on tracing wakeup events is still very accurate up to a load of 45%, and keeps remaining generally more accurate than

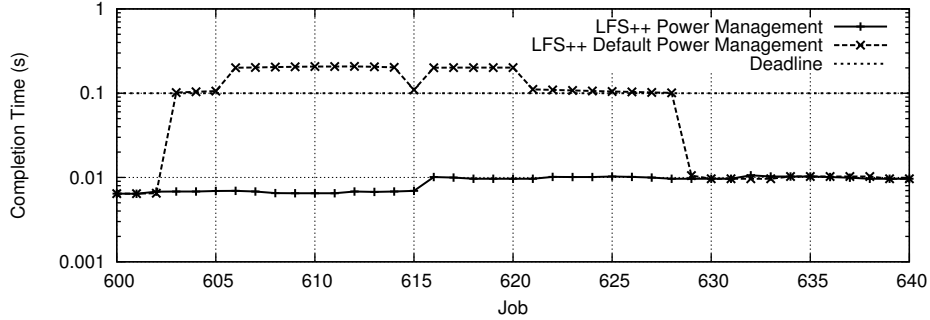


Fig. 9. Job completion times of a periodic application controlled by LFS++ when using the integrated power management logics, as compared to when using the default one.

system call tracing (up to a load of 85%), as shown by the respective standard deviation curves.

7.4 Power Management

To understand the need for integrating power management into LFS++ we have observed the behaviour of a real-time periodic application (with a period of 100 ms), executed on a system with frequency scaling supported and enabled. We ran `rt-app` using first the LFS++ power management logic, then disabling it, but enabling an external instance of `powernowd`.

Figure 9 shows the completion times of the jobs of the application during a frequency transition. Before job 600 the system is executing a background best effort workload (a simple cpu hog), that causes both of the power management logics (the one embedded in LFS++ and the default one) to maintain the system at its highest frequency. When the background workload is stopped, the total load of the system decreases, and both of the power management logics decrease the frequency of the CPU. The CPU frequency increases the response time approximately in proportion to the ratio between the old and the new frequency (resp. 1.5 GHz and 2.4 GHz). In the case where LFS++ is not synchronised with the power management mechanism, the increase in response time is not reflected soon enough in an increase of the budget assigned to the application; this causes the overruns shown in the figure, that last until job 629; at that point the feedback logic adapts itself to the new computation time of the application and we see no more overruns.

When LFS++ acts also as the power manager in the system, then we see no overrun, because before updating the CPU speed LFS++ waits for the completion of the mode-change protocol (from the figure we can see that the actual CPU speed change happens some time after than in the other case), and the new budget is sufficient to fit the increased execution time.

7.5 LFS++ Overhead

The technique presented in this paper requires a run-time mechanism to be flanked to the real-time applications in the system, which constantly monitors them and adapts the scheduler configuration. This implies a set of run-time overheads that

Tracer	Average (sec)	Relative average	Standard deviation (sec)
NOTRACE	14.834	-	1.04432
QTRACE (WAKEUPS)	14.837	0.02%	1.04843
QTRACE (SYSCALLS)	14.945	0.74%	1.11884
QOSTRACE	15.156	2.17%	1.05422
STRACE	15.727	9.32%	0.796605

Table II. Overhead introduced by various tracers, compared to when no tracer is used (first row).

can be summarised as follows:

tracing overhead. due to the tracing mechanism, which needs to record the set of interesting events for the traced applications;

period detection overhead. due to the Fourier transform computation and analysis needed to infer the period of the traced applications;

adaptive scheduling overhead. due to the adaptation loop which LFS++ embraces for each traced application, in which the workload in the last sampling instant is observed and the reservation budget is accordingly modulated.

7.5.1 Tracing Overhead. The tracing overhead has been evaluated by measuring the time spent by `ffmpeg`⁴ to transcode a video, with various system-call tracers attached during the entire run. Each run has been repeated 10 times, and the average and standard deviation of the total transcoding time have been computed. Results are reported in Table II. First, we determined a baseline, running the transcoding process without any tracer active, then we traced the program with our `qtrace` tracer, described in Section 6.2.

The measured overhead includes both the time for logging the system-call information within the kernel, which is really negligible and hard to measure, and the one needed by `lfs++` to download the time stamps through a special device, which introduces a few context switches towards the tracing process (much fewer than when using `ptrace()`-based tools). Finally, for completeness, also the overhead obtained while tracing the same program by using the standard `strace` Linux tool and the `qostrace` tool presented in [Cucinotta et al. 2009] are reported. As it can be seen, the presented tracer, when tracing system calls exhibits an overhead close to 0.7% (relative to the application computation time), and almost no overhead (under the measurement noise threshold) when tracing wakeups.

7.5.2 Period Detection. The period detection overhead is due to various components, such as the computation of the Fourier Transform and the Peak Detection Heuristic. Hence, the time needed for period detection depends on the number of generated events (which in turn depends on the observation interval horizon) and on the target frequency range and granularity, as discussed in Section 5.2.1. Section 7.3 already presented an experimental evaluation of the impact of the observation interval on the number of generated events and on the precision of the rate

⁴More information is available at <http://www.ffmpeg.org>.

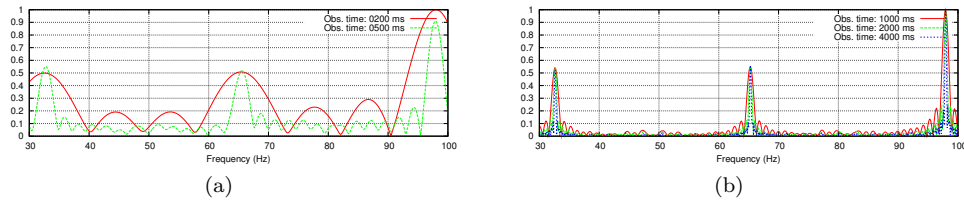


Fig. 10. Normalized frequency-transform of the events obtained by tracing `mplayer` at varying tracing time.

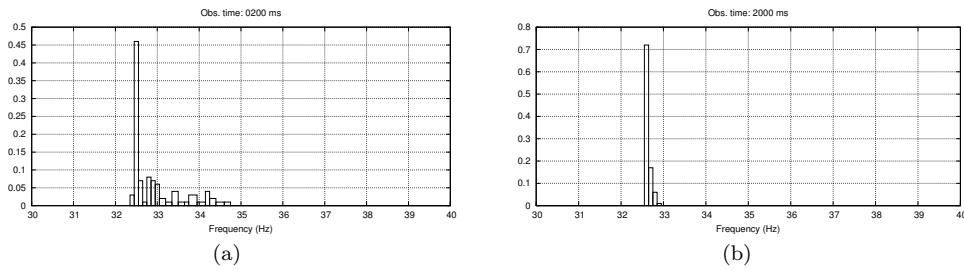


Fig. 11. PMF of the frequency detected by LFS++ for `mplayer` at varying tracing times.

detection mechanism. The results presented in such a section are consistent with another, more extensive, evaluation of the overheads that has already appeared in our previous work [Cucinotta et al. 2010].

Here, a short summary of the experiments presented in the previous paper (based on `mplayer` reproducing a set of mp3 files) is reported. Figure 10 presents the plot of the amplitude spectrum obtained for different tracing intervals (note that in order to enhance readability, values on the Y axis have been normalised to the maximum value of the amplitude spectrum - hence the highest peak is 1.0). As the plots in Figure 10 (a) show, the periodic nature of the application is evident already from a tracing time of $500ms$, in which the peaks of the curve close to the 32.5 , 65 and $97.5Hz$ frequencies are quite evident. However, the plots in Figure 10 (b) show that the periodicity becomes indisputable starting from $1s$ of tracing time, and beyond.

Each operation of tracing and period-detection with a given tracing time has been repeated 100 times, and the PMF curves of the detected frequency have been computed and reported in Figure 11. In Figure 11 (a), it is shown that a tracing time as short as $200ms$ may lead to a small error in the detected frequency, that remains between $32.5Hz$ and $35Hz$ most of the time, with a few occurrences on the second harmonic at $97.5Hz$ (not shown on the plots for enhancing readability). Increasing the tracing time, the PMF becomes tighter around the $32.5Hz$ value, however the relatively few occurrences (between 0 and 2 on the 100 repetitions) of the second harmonic persist.

8. FUTURE WORK AND CONCLUSIONS

In this paper, we have proposed an adaptive mechanism for real-time scheduling of periodic legacy applications. Our first contribution is to show (by theoretical and experimental data) that an effective choice for the scheduling parameters can be made only based on a correct estimation of the activation period and of the computation time of the task. Our second contribution is to show algorithmic solution to estimate these parameters based on a trace of events generated in the kernel. The solution we outline has been implemented in the Linux kernel and we show, as our third contribution, how to tackle the architectural issues that the approach raises. Finally, we offer full evidence of the effectiveness of the approach on a large collection of experimental data, which displays the radical improvement in performance over the standard scheduling solutions.

Our future investigation will take two different directions. The first one is the adaptation of the mechanism to the case of multi-threaded applications. The results that we collected are encouraging but the technique needs some refinement (e.g., for the evaluation of several periods from a single trace). The second one is the extension of the technique to symmetric multi-core machines. In this context, an open research issue is to design an optimised cooperation between the load balancing mechanisms inside the kernel, the real-time partitioning of the tasks between the cores and the adaptive mechanisms proposed in this paper.

REFERENCES

- ABENI, L. AND BUTTAZZO, G. 1998. Integrating multimedia applications in hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*. Madrid, Spain.
- ABENI, L. AND BUTTAZZO, G. 1999. Adaptive bandwidth reservation for multimedia computing. In *Proceedings of the IEEE Real Time Computing Systems and Applications*. Hong Kong.
- ABENI, L., CUCINOTTA, T., LIPARI, G., MARZARIO, L., AND PALOPOLI, L. 2005. Qos management through adaptive reservations. *Real-Time Systems Journal* 29, 2-3 (March).
- ABENI, L. AND PALOPOLI, L. 2008. Adaptive real-time scheduling for legacy applications. In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2008)*. Hamburg, Germany, 583–590.
- ABENI, L. AND PALOPOLI, L. 2009. Legacy real-time applications in a reservation-based system. *IEEE Transactions on Industrial Informatics* 5, 3 (August).
- ABENI, L., PALOPOLI, L., LIPARI, G., AND WALPOLE, J. 2002. Analysis of a reservation-based feedback scheduler. In *Proc. of the Real-Time Systems Symposium*. Austin, Texas.
- BOGERT, B., HEALY, M. R., AND TUKEY, J. 1963. The quefrency analysis of time series for echoes: Cepstrum, psuedo-autocovariance, cross-cepstrum and saphe cracking. In *Proceedings of the Symposium on Time Series Analysis*, N. Wiley, Ed. 209–243.
- C. LU, J. STANKOVIC, G. T. AND SON, S. 2002. 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).
- CASIMIRO, A. AND VERISSIMO, P. 2001. Using the timely computing base for dependable qos adaptation. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*. New Orleans, Louisiana.
- CUCINOTTA, T., ABENI, L., PALOPOLI, L., AND CHECCONI, F. 2009. The wizard of os: a heartbeat for legacy multimedia applications. In *Proceedings of the 7th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia 2009)*. Grenoble, France.
- CUCINOTTA, T., CHECCONI, F., ABENI, L., AND PALOPOLI, L. 2010. Self-tuning schedulers for legacy real-time applications. In *Proceedings (to appear) of the European Conference on Computer Systems (Eurosys 2010)*. European chapter of the ACM SIGOPS, Paris, France.

- EIDE, E., STACK, T., REGEHR, J., AND LEPREAU, J. 2004. Dynamic cpu management for real-time, middleware-based systems. In *Proceedings of the 10th IEEE Real Time Technology and Applications Symposium (RTAS 2004)*. Toronto, Canada.
- GOEL, A., WALPOLE, J., AND SHOR, M. 2004. Real-rate scheduling. In *Proceedings of the 10th IEEE Real Time Technology and Applications Symposium (RTAS 2004)*. Toronto, Canada.
- IEEE. 2004. *Information Technology -Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) Amendment: Additional Realtime Extensions*.
- JONES, M. B., MCCULLEY, D. L., FORIN, A., LEACH, P. J., ROSU, D., AND ROBERTS, D. L. 1996. An overview of the rialto real-time architecture. In *Proceedings of the 7th ACM SIGOPS European Workshop*. Connemara, Ireland.
- KEDEM, B. 1986. Spectral analysis and discrimination by zero-crossings. *Proceedings of the IEEE* 74, 11 (Nov.), 1477 – 1493.
- KRASIC, C., SAUBHASIK, M., SINHA, A., AND GOEL, A. 2009. Fair and timely scheduling via cooperative polling.
- KRISHNAMURTHY, Y., KACHROO, V., KARR, D. A., RODRIGUES, C., LOYALL, J. P., SCHANTZ, R. E., AND SCHMIDT, D. C. 2001. Integration of QoS-enabled distributed object computing middleware for developing next-generation distributed application. In *LCTES/OM*. 230–237.
- LEHOCZKY, J., SHA, L., AND DING, Y. 1989. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of the Real Time Systems Symposium*. 166–171.
- LESLIE, I. M., MCAULEY, D., BLACK, R., ROSCOE, T., BARHAM, P., EVERS, D., FAIRBAIRNS, R., AND HYDEN, E. 1996. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications* 14, 7.
- LI, B. AND NAHRSTEDT, K. 1998. A control theoretical model for quality of service adaptations. In *Proceedings of Sixth International Workshop on Quality of Service*.
- LIU, C. L. AND LAYLAND, J. 1973. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM* 20, 1.
- MERCER, C. W., RAJKUMAR, R., AND TOKUDA, H. 1993. Applying hard real-time technology to multimedia systems. In *Workshop on the Role of Real-Time in Multimedia/Interactive Computing System*.
- MOK, A. K., FENG, X., AND CHEN, D. 2001. Resource partition for real-time systems. In *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium*. Taipei, Taiwan, 75–84.
- PALOPOLI, L., CUCINOTTA, T., MARZARIO, L., AND LIPARI, G. 2009. AQuoSA — adaptive quality of service architecture. *Software – Practice and Experience* 39, 1, 1–31.
- PISZCZALSKI, M. AND GALLER, B. A. 1979. Predicting musical pitch from component frequency ratios. *The Journal of the Acoustical Society of America* 66, 3, 710–720.
- RAJKUMAR, R., JUVVA, K., MOLANO, A., AND OIKAWA, S. 1998. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*.
- STEERE, D. C., GOEL, A., GRUENBERG, J., MCNAMEE, D., PU, C., AND WALPOLE, J. 1999. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. New Orleans, Louisiana, USA, 145–158.
- TSETSEKAS, C. A., MANIATIS, S., AND VENIERIS, I. S. 2001. Supporting qos for legacy applications. In *ICN. Lecture Notes in Computer Science*. Springer, 108–116.
- VERISSIMO, P. AND CASIMIRO, A. 2002. The timely computing base model and architecture. *IEEE Transactions on Computers* 51, 8 (August).
- WÜST, C. C., STEFFENS, L., VERHAEGH, W. F., BRIL, R. J., AND HENTSCHEL, C. 2005. Qos control strategies for high-quality video processing. *Real-Time Syst.* 30, 1-2, 7–29.
- YANG, T., LIU, T., BERGER, E. D., KAPLAN, F., AND MOSS, J. E. B. 2008. Redline: First class support for interactivity in commodity operating systems. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*. San Diego, CA.