Self-tuning Schedulers for Legacy Real-Time Applications *

Tommaso Cucinotta Fabio Checconi
RETIS – Scuola Superiore Sant'Anna
Pisa, Italy
cucinotta@sssup.it fabio@gandalf.sssup.it

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

Abstract

We present an approach for adaptive scheduling of soft real-time legacy applications (for which no timing information is exposed to the system). Our strategy is 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. By a thorough experimental evaluation of an implementation of our approach, we show its performance and its efficiency.

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

General Terms Experimentation, Performance, Measurement

1. Introduction

We focus on soft real-time applications for which occasional violations of the timing constraints are acceptable anomalies as far as they are under control. Multimedia streaming is a perfect example of this kind. The compliance with temporal constraints of an application like this in multi-task systems is a challenging problem. The most effective strategies require a converging effort from application developers and operating system designers. The operating system has to provide applications with an adequate support in terms of scheduling algorithms and of resource management policies. The application developer has to use the real-time mechanisms of the kernel through a specialised API and to appropriately select the scheduling parameters to enforce timing constraints.

The scheduling support for real-time applications, in general purpose operating systems (GPOS), is typically limited to fixed priorities, known to be unfit for soft real-time applications. A better alternative is offered by such soft real-time schedulers as the *resource reservations* [24], available in real-time variants of the Linux kernel and in other real-time operating systems (RTOS). These algorithms ensure a correct temporal partitioning of the system resources whereby each application is guaranteed a fraction of the CPU time. Roughly speaking, if one knows the timing parameters of a task, it is possible to dimension the reservation parameters to achieve a given probability of a deadline miss [1] or prevent deadline misses altogether. When application requirements are scarcely known or time-varying, an interesting possibility is to adapt the scheduling parameters while the application runs [3, 4]. The idea is that if an application is structured as a (typically periodic) stream of jobs and if it notifies by appropriate API calls the start and termination of each job, a feedback controller can be used to monitor the difference between the actual execution of the job and its temporal constraints and take corrective actions as needed (i.e., to increase the reserved CPU time in presence of delayed executions or to reduce it for early terminations). An API of this kind is the one developed for the AQuoSA project[1].

The use of a specialised API is relatively easy for applications developed from scratch. When the source code of the application is available, it is possible to review the code inserting the appropriate API calls. This re-factoring is not effortless and software producers are not often inclined to take the risk of this development. In other cases, the source code of the application is simply unavailable. In this paper, we use the term legacy applications with reference to *applications that are characterised by some temporal constraints, but are not developed using a specific API*. Therefore, developers of legacy applications contrive (or contrived) to achieve an acceptable timing behaviour by a large range of heuristic solutions (including a generous use of buffering). The robustness of this solution (and often its responsiveness) is, in this way,

* The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7 under grant agreement n. 214777 "IRMOS—Interactive Realtime Multimedia Applications on Service Oriented Infrastructures", and under grant agreement n. IST-2008-224428 "CHAT—Control of Heterogeneous Automation Systems".

EuroSys'10, April 13–16, 2010, Paris, France.

[1] The project website: `http://aquosa.sourceforge.net`

at a serious risk of being compromised. In contrast, we make the point that *even for legacy applications the best way to obtain an acceptable timing behaviour is by operating at the scheduling level*. The greatest obstacle along this way is how to design an effective policy for the selection of the scheduling parameters. Indeed, when dealing with legacy applications, we do not know the exact timing requirements associated with each task in advance because the information on the structure of the application is unavailable or difficult to reconstruct. Neither are we able to infer these requirements adaptively because the application does not use specific API calls that identify the start and the end of a job [2].

In this paper, we propose a comprehensive solution to the problem of real-time scheduling of legacy applications that develops and substantiates a preliminary idea presented in [8]. Our approach is: 1) to infer as much information as possible on the timing requirements of the application from the "black box" observation of the kernel events it generates, and 2) to adapt the scheduling parameters online from measurements related to the real-time behaviour of the application. More in detail, the first contribution that we report in the paper is a kernel-level tracing mechanism, which records the events generated by each task. This mechanism is not intrusive: it introduces a negligible overhead and it can be used for any legacy application (for instance, it does not require the use of a debugger that would breach the license of some applications). The second contribution is the design of an event analyser based on signal processing theory that identifies the activation parameters of the tasks (i.e., the periods) and, hence, their timing requirements. In order for the tasks to be able to fulfil these requirements, it is required that they receive an allocation of CPU time sufficient to satisfy their computation request in due time. The third contribution is then a feedback scheduler that identifies the bandwidth requirements based on the temporal behaviour of the task. Since the application does not pro-actively supply any information on its deviation from the ideal timing behaviour, the feedback scheduler performs an indirect assessment of this quantity by sampling the state variables of the scheduler. The whole machinery has been implemented in the Linux kernel and has been validated extensively applying the framework to a variety of legacy multimedia applications.

The paper is organised as follows. Section 2 reviews the related work in the literature. Section 3 introduces the problem of identifying optimum scheduling parameters for legacy multimedia applications. Section 4 presents the general proposed methodology for addressing the problem, while Section 5 presents its experimental evaluation conducted on a Linux-based implementation. Section 6 contains a road-map for further research on the topic, along with a few concluding remarks.

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 [1, 21, 24] have been proposed. Such algorithms enable a fine-grained control on the CPU 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 in 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. 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 [3–5, 12, 18]. In such approaches, while the applications execute, their real-time behaviour is monitored and corrective actions are taken changing the scheduling parameters so that specified QoS related objectives are met.

Computing models that represent an alternative to the real-time tasking model have been proposed by different authors. An interesting example is offered by the Timely Computing Base - TCB - model proposed by Verissimo et al. [28]. The authors proposed an interesting combination of the TCB with an application-level QoS adaptation [6] 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 [13, 14, 17].

A piece of work that bears some resemblance with this paper is the one proposed by Steere et.al. [26], 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. [10], in the context of the QuO framework [15]. 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 in entirely transparent to the applications.

The problem of providing QoS guarantees for legacy applications has been also explored in the networking community. Tstetekas et al. [27] 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 [29], 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) [2]. In the LFS scheme, the scheduler samples for each task 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 produces a better QoS.

As far as the problem of reconstructing the task period is concerned, important reference points are the approaches developed in the literature of digital processing of sound signals, where different approaches have been developed to extract the pitch and identify the fundamental frequency [11, 20]. Such techniques served as a good starting point for our analyser, but we had to adapt them to the analysis of a time-series of events.

3. The Problem

In this paper, we are concerned with legacy applications that do have real-time requirements but do not possess a structure that makes them fit into the classical real-time tasking model. Before going into the details of the specific issues related to this kind of applications, it is useful to provide some background information on the real-time tasking model and on the scheduling algorithm that underlies our work.

3.1 Background

The Real-Time Tasking Model. In the real-time scheduling 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.

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 T_i^s. 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 the Constant Bandwidth Server (CBS) [1], 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 an EDF policy. The reader is referred to the cited paper for a longer discussion.

3.2 Selecting the Scheduling Parameters

When we use a reservation-based scheduler to schedule a real-time task, the problem arises of how to choose the scheduling 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 we use a CBS to schedule a periodic task having period P_i and if we know its worst case execution time C_i, we can simply set $T_i^s = P_i$ and $Q_i^s = C_i$ and the task provably meets all of its deadlines [1]. Alternatively, if we know the distribution of the inter-arrival and execution times, the server parameters T_i^s and Q_i^s can be set so that the task misses its deadlines with a given probability. If a single server is used to schedule multiple tasks, hierarchical scheduling analysis [22] 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. A tentative choice of the parameters can lead to severe malfunctioning of the application. 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

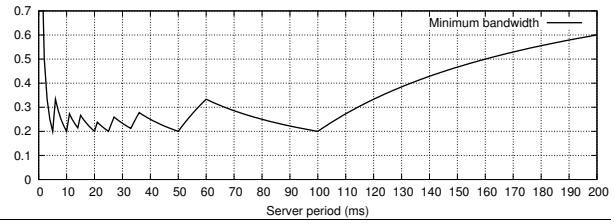


Figure 1. Fraction of CPU Q_i^s/T_i^s required to correctly schedule a real-time task with 20% utilisation $C = 20$ ms, $P = 100$ ms.

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 Quality of Service. Likewise, choosing a large value of Q_i^s affects adversely the behaviour of the other applications and 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 [8] using an analysis technique inspired to the supply bound function [16]. 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 $C_i = 20$ms, $T_i = 100$ms. As it is possible to see in Figure 1, the required bandwidth ranges from the correct value (20%) to very 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, $T_i^s = \frac{P_i}{3} = 33$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.

If we schedule multiple tasks in the same server, things are far less obvious. This choice has natural motivations if we use the CBS to schedule a multi-task application or to implement a machine virtualisation scheme with performance guarantees but it raises important issues as well. As an illustrative example, consider a task-set composed of three real-time tasks with parameters: $C_1 = 3.0$ms, $P_1 = 15.0$ms, $C_2 = 5.0$ms, $P_2 = 20.0$ms, $C_3 = 5.0$ms, $P_3 = 30.0$ms. Suppose that the three tasks are scheduled in the same reservation and, inside the reserved time, the allocation is decided using a fixed priority schedule. The priorities are chosen proportionally to the activation rate, the famous Rate Monotonic assignment [19]. Applying the theory of hierarchical scheduling [9, 22, 25], we are able to identify, for each server period, the minimum budget to ensure the respect of timing constraints (and hence the bandwidth). We show this minimum bandwidth in Figure 2. For the reader convenience, we report in the same plot the cumulative utilisation of the three tasks. The figure lends itself to the following considerations: 1) in this case, there is not an obvious connection between the "best" server period and the periods of the tasks, 2) even with the best choice of the service period the efficiency is way below the one that we can get with a separate server for each thread (62%). Indeed, with a single reservation the waste of bandwidth is between 6% and 41%. On the contrary, if we schedule each task in a dedicated server and if the period of the tasks is correctly identified, we can schedule the three tasks with a total assignment of bandwidth equal to their cumulative utilisation, the theoretical lower bound.

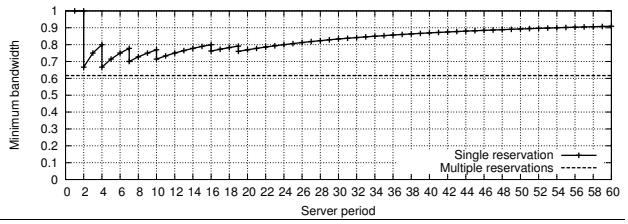


Figure 2. Minimum bandwidth required to schedule in a single reservation three tasks. The task parameters are $C_1 = 3$ms, $P_1 = 15$ms, $C_2 = 5$ms, $P_2 = 20$ms, $C_3 = 5$ms, $P_3 = 30$ms and the cumulative utilisation is $\approx 62\%$.

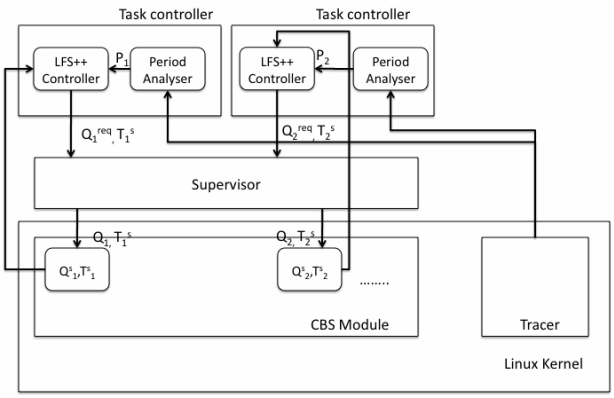


Figure 3. Scheme of the proposed approach.

4. Our Approach

The approach proposed in this paper is pictorially described in Figure 3. The legacy real-time tasks are scheduled through a CBS scheduling mechanism implemented in the Linux kernel. A *task controller* is associated with each CBS server to the purpose of identifying the correct parameters (Q_i^s, T_i^s) for the task scheduled in the server. More specifically, the controller formulates a request for a couple of parameters Q_i^{rep}, T_i^s. The request is submitted to the *supervisor* component whose purpose is to enforce the schedulability con-

dition

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

Namely, if the requests from the task controllers do not saturate the total available bandwidth, requests can be entirely granted $Q_i^s = Q_i^{req}$. Otherwise they have to be curbed to fit in the bound. More information on the supervisor, along with implementation details, can be found in [23]. From now on, we focus on how to design the task controllers for legacy applications.

The task controller is activated periodically and is composed 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 (feedback controller) samples the state of the scheduler to compute the CPU time utilised by the application during the last sampling period. This information is combined with the estimated period to identify a correct pair of reservation parameters.

The design of the tracer mechanism, of the feedback controller and of the period analyser is a challenging design activity tapping different disciplines. In this section, we will discuss the most important theoretical and architectural issues underlying each one of these components.

4.1 System Call Tracer

A periodic task typically 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 block itself, we could in principle trace its execution instants and evaluate the interval of time between two subsequent calls to estimate the period. 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 all the system calls generated by the application. As an example, in Figure 4, we show a statistic of all the different system calls recorded for a three minutes execution of `mplayer` reproducing a video file. Most calls are `ioctl()` calls, which are needed for dealing with the audio device via the Linux `ALSA` sound subsystem (through the `libasound` library).

From this very complex sequence of events, we extract the dominant periodic pattern using the algorithm illustrated in Section 4.2. In this section, our primary concern is to show how to perform the tracing operation minimising the overhead incurred in recording the events.

A standard tool used in Linux to perform the tracing operation is `strace`, based on the `ptrace()` system call. This tool was mainly developed for debugging purposes and the overhead can be unsustainable in a "production" use. In our previous work [8] we proposed an alternative to `strace`, called `qosstrace` that features, in our evaluation, a remarkable overhead reduction w.r.t. `strace`. However,

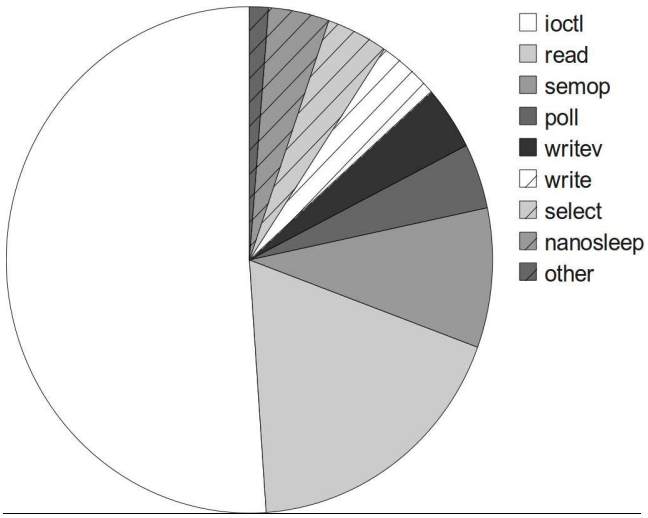


Figure 4. Statistics of the system calls performed by `mplayer`

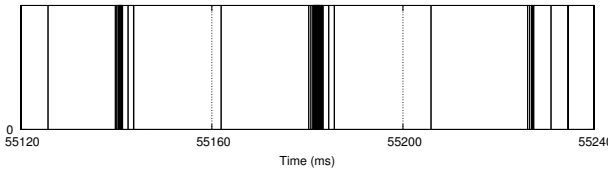
even `qostrace` suffers an important limitation inherent to the very use of `ptrace()`. Indeed, using `ptrace()`, the monitored process is blocked on each system call. The tracer process is woken up to inspect the context of the monitored process (or to get the current time) before returning the execution to the monitored process. Hence, the system has to execute two context switches whose duration is a lower bound for the overhead of any solution based on `ptrace()`.

To overcome this limitation, we introduce here a novel solution based on two components:
1) a kernel patch, which, upon entry and exit points of a system call at the kernel-level, records the timestamps associated with the start and the termination of the call;
2) a user-space program, which, operating through a character device, downloads a batch of time instants associated to the system call executed in the last sampling period and forwards it to the period analyser.

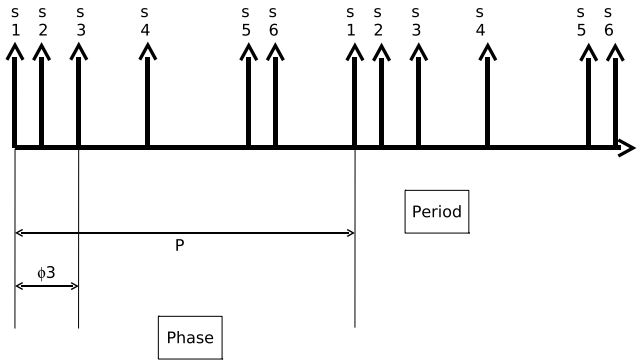
The data structure used to log the timestamps is a statically allocated circular buffer. The kernel patch can selectively trace a specified subset of system calls for a specified subset of running processes. This information is passed to the patch through the character device. In this way, it is possible to avoid the tracing of system calls that are totally unrelated with the scheduling events (avoiding unnecessary noise for the analyser) and the use of too large buffers.

4.2 The Period Analyser

To reconstruct the period, we make the reasonable assumption that the real-time application generates periodic bursts of system calls 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 that this assumption is well founded, consider the excerpt of a trace recorded for a real application and reported in Figure 5.(a), where each event is represented as a vertical line. As it is possible to see most of



(a)



(b)

Figure 5. A) a sequence of events associated to a segment of execution of an application, B) The mathematical model as a sequence of Dirac Deltas (δ).

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 (system call) with a Dirac delta (δ). Therefore, if s_i symbolically represents a system call (e.g., `clock_nanosleep`), the sequence of events associated with this call 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 call event inside the period. 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 system calls called by the task. An example of a trace that adheres to this pattern is displayed in Figure 5.(b). 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 the signals is:

$$S(\omega) = \mathcal{F}(s(t)) = \tfrac{2\pi}{P} \sum_{i=1}^{K} \sum_{n=-\infty}^{\infty} e^{-jn\omega_0\phi_i} \delta(\omega - n\omega_0).$$

where, $\omega_0 = 2\pi/P$.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:

$$S(\omega) = \tfrac{2\pi H}{P} e^{-j\omega H/2} \sum_{i=1}^{K} \sum_{n=-\infty}^{\infty} e^{-jn\omega_0\phi_i} \text{sinc}((\omega - n\omega_0)\frac{H}{2}). \quad (2)$$

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

The equation above consists of a sum of complex vectors with amplitude $\text{sinc}((\omega - n\omega_0)\frac{H}{2})$ and phase given by the direction of the complex number $e^{-jn\omega_0\phi_i}$. 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 $\omega = n\omega_0$. Hence, the distance between $\omega_0 = 2\pi/P$ 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.

4.3 Computation of the spectrum

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

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

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

This very simple approach is much more convenient than the application of algorithms computing the Fast Fourier Transform (FFT). Indeed, the latter requires the specification of a sampling time that in our case should be very small (in the order of nanoseconds) because events can take place at any point in time and are recorded with a very high precision in the kernel. The resulting signal would be null most of the time. Hence, the computation of the FFT would be utterly inefficient.

4.3.1 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 $S(\omega)$ of the signal $s(t)G_H(t - H/2)$ (the modulus of its Fourier Transform) in the frequency range $[\omega_{min}, \omega_{max}]$, with

step $\delta\omega$, as discussed above:

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

2. identify a first set of peaks $\omega_1, \ldots, \omega_m$ as the local maxima of the amplitude spectrum in the range (ordered by frequency);
3. discard all peaks ω_i for which $S(\omega_i)$ is lower than α times its average value $\overline{S}$ (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 ω_i, compute the sum Σ_i of the amplitude spectrum in correspondence of at most k^{max} integer multiples of ω_i, (set to 10 in the experiments) with a tolerance of ϵ, i.e.,

$$\Sigma_i = \sum_{\substack{\omega_j \in [h\omega_i - \epsilon, h\omega_i + \epsilon] \\ j \in 1,\ldots,10, \omega_j \leq \omega_{max}}} |S(\omega_j)|.$$

6. select the frequency ω_i with 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 then 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 10 (to prevent secondary peaks from outweighing the main one due to their high number).

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 $F \triangleq \frac{\omega_{max} - \omega_{min}}{\delta\omega}$ be the number of computed samples for $|S(\omega)|$. The second and the third steps of the algorithm require the analysis of all the samples. Then (step 5), for each candidate peak frequency ω_i, the values of the transform in correspondence of the integer multiples of ω_i, with a tolerance of ϵ, are summed up, up to ω_{max}. The number of sums to make is given by $\min\left\{\frac{\omega_{max}-\omega_i}{\omega_i}, k^{max}\right\}\frac{\epsilon}{\delta\omega}$; 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:

$$E = \frac{\omega_{max} - \omega_{min}}{\delta\omega} + \sum_{\omega_i \in F_{max}} \min\left\{\frac{\omega_{max} - \omega_i}{\omega_i}, 10\right\}\frac{\epsilon}{\delta\omega}, \quad (5)$$

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

4.4 LFS++

The purpose of the LFS++ controller is to estimate the CPU utilisation of the task and assign the bandwidth accordingly based on periodic measures of the computation time of the task. To this end, we require the presence of an appropriate "sensor" inside the kernel that measures the CPU time consumed by the application in each interval. This information may be fed to a "predictor" or "estimator" component which may easily determine what is the budget that best suites the application needs, based on the observation of past computation times of the application. 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 or of the thread. In our specific case, we used the API of the AQUOSA middleware and in particular the system call `qres_get_time()`, which returns the CPU time executed by a thread attached to a CBS, starting from a specified time in the past.

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 to be used in the next sampling interval is determined as follows:

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

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/P is the bandwidth requested by the controller). The predictor $\mathcal{P}$ can be implemented in different ways. In this paper, we propose a "quantile estimator", which basically takes a set of past observed N samples, and outputs the estimated p^{th} quantile of the computation times distribution. This may be easily

accomplished for π values which are expressed as $\frac{N-j}{N}$, where j is an integer. For example, with $N = 16$, if $p = 1.0$ then one has to take the maximum over the last N samples. For $p = 0.9375$ one has to take the second maximum, and so forth. A few remarks are in order:

1. This mechanism does not actuate a "punctual" control on the timing behaviour of each job. Indeed, assuming even that the predictor returns the maximum of the past N samples, the control law actually sets the bandwidth to the maximum *average* utilisation time experienced by S/T consecutive jobs over the last N observations. This policy corresponds to a good job-wise bandwidth assignment *if* the computation time remains uniform over the job. For a workload such as the one generated by an MPEG video application with a fixed group of pictures that generates periodic computation peaks for the I frames and a much lower computation time for P and B frames, this policy could determine delays for the job corresponding to the I frames decoding. We conjecture that a closer cooperation with the scheduler for detecting budget exhaustion might help us cope with this issue and provide this specific class of application with a better support.
2. One might be tempted to set the sampling period S to the estimated period P in order to perform a job-wise adaptation. In fact, the result of such a choice would be disappointing since the feedback would anyway operate asynchronously w.r.t. the job release instants. Due to the random interference undergone by the task, such a choice simply determines a very unstable and fluctuating behaviour for the predicted computation time with no apparent benefit.
3. 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 increase the responsiveness of the system to changes in the workload.

5. Experimental Results

An extensive experimental evaluation of the approach has been performed, we using the Linux kernel `2.6.29` series, modified so as to include the `AQuoSA` real-time scheduler [23], the `qtrace` kernel-level tracer described in Section 4.1, and a user-space application called `lfs++` implementing the spectrum analysis and feedback-based scheduling algorithm itself. Furthermore, we used an implementation of the CBS in order to compare with our previous LFS approach appeared in [2]. The machine used for the tests is based on an Intel(R) Core(TM) 2 Duo CPU at $2.6\,GHz$, with an operating frequency fixed at $800\,MHz$, running an Ubuntu 9.04 Linux Operating System.

Tracer	Average (sec)	Relative average	Standard deviation (sec)
NOTRACE	21.0916	-	0.094951
QTRACE	21.2253	0.63%	0.143581
QOSTRACE	21.658	2.69%	0.221327
STRACE	22.2536	5.51%	0.140593

Table 1. Overhead introduced by various tracers, compared to when no tracer is used (first row).

In the first set of experiments reported in this section, we highlight the overhead of our technique and the corresponding period analyser accuracy with respect to the available parameters, and the real-time load possibly present on the system. In a second set of experiments, we show the effectiveness of the approach in providing scheduling guarantees to multimedia applications by adopting an application-level QoS metrics, i.e., the inter-frame times for a video player.

Many of the experiments have been performed by using `mplayer`[2], a popular media player for Linux. However, the obtained results and especially the capability to extract the period have been verified also on various other players always on Linux, including (details omitted): `vlc`, `realplayer`, `sox`. When an application-level QoS measurement was needed, we used a custom video pla which records the sequence of inter-frame times.

5.1 Period Analyser Overhead

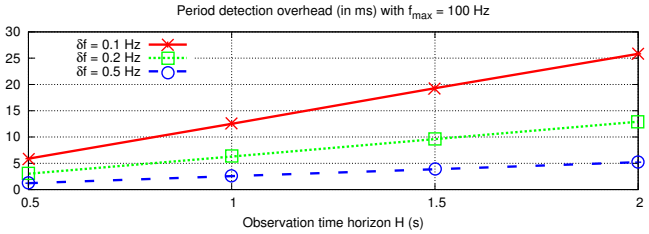
Tracing overhead. The tracing overhead has been evaluated by measuring the time spent by `ffmpeg`[3] to transcode a video, with various system-call tracers attached during the entire run. Each run has been repeated 10 times, and the average and standard deviation of the total transcoding time have been computed. Results are reported in Table 1. 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 4.1.

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 [8] are reported. As it can be seen, the new presented tracer exhibits an overhead close to 0.6%, relative to the application computation time, far lower than the others.

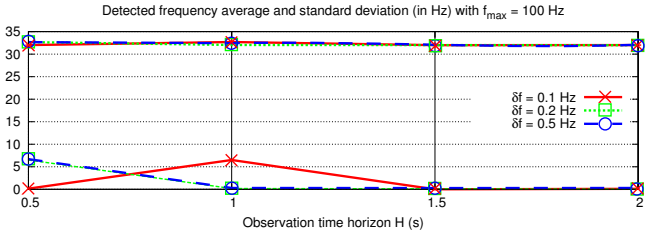
Fourier Transform Overhead. The overhead due to the Fourier transform computation is now evaluated. In the dis-

[2] More information is available at `http://www.mplayerhq.hu`.

[3] More information is available at `http://www.ffmpeg.org`.

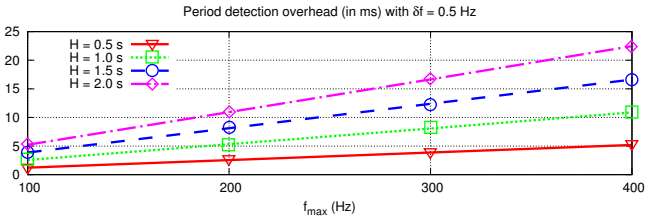


(a)

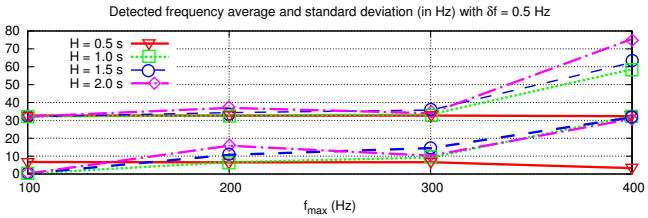


(b)

Figure 6. Time to compute the frequency transform (top), and corresponding precision in frequency detection (bottom), as a function of the observation time H and δf, at fixed $f_{max} = 100Hz$ and $\epsilon = 0.5Hz$.



(a)



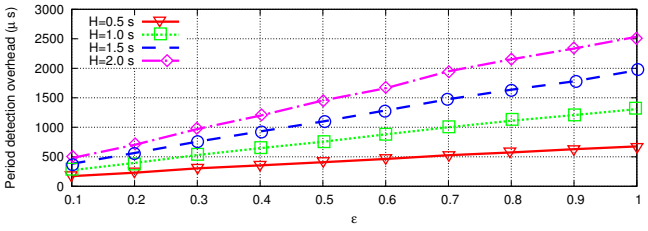
(b)

Figure 7. Time to compute the frequency transform (top), and corresponding precision in frequency detection (bottom), as a function of the observation time H and f_{max}, at fixed $\delta f = 0.5Hz$ and $\epsilon = 0.5Hz$.

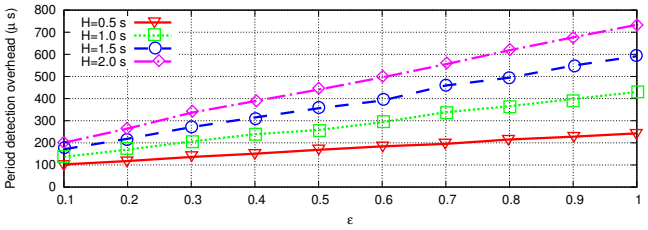
cussion we use the frequency f (in Hz) instead of the variable $\omega = 2\pi f$ used in the previous section (expressed in rad/s). Figures 6 (a) and 7 (a) shows the time needed to compute the spectrum, as a function of the available parameters. The shown values are averaged through 100 executions of the algorithm while tracing the `mplayer` application playing an mp3 song. The experiments confirm the theoretical expectations of Equation (3), the transform computation time being proportional to both the number of detected events (which is in turn proportional to the observation time horizon H), and to the number of frequency values in which the transform is sampled (which is equal to $\frac{f_{max}-f_{min}}{\delta f}$).

In Figures 6 (b) and 7 (b) we report the variability of the period analyser result, as a function of the same parameters. In Figure 6 (b), we can see that the detected frequency and its precision is not affected sensibly by increasing the δf from $0.1\,Hz$ to $0.5\,Hz$, which, on the contrary, has a big impact the computation overhead, as displayed in Figure 6 (a). Also, in Figure 7 (b) we can see that by increasing f_{max} we generally increase the variability of the detected frequency,

Period Detection Heuristic. In Figure 8, the time needed to extract the period from the given frequency transform is shown. The measurement was repeated 100 times over different event sets coming from the same traced program under the same conditions, and the average and standard deviation values have been computed over the repetitions. Figure 8 (a) corresponds to the heuristic trying all the possible frequencies, while Figure 8 (b) uses the α threshold described at the step 3 of the algorithm presented in Section 4.3.1, set to $\alpha = 20\%$. The pictures highlight that the measured overhead is basically in linear relationship with both the observation time H and the ϵ parameter, as foreseen in Equation (5).



(a)



(b)

Figure 8. Period detection overhead, as a function of the observation time H and the ϵ parameter.

Also, by comparing the top and bottom plots, it is possible to appreciate the overhead decrease due to the cut of the local candidate peaks due to the α threshold.

In Figure 9, we report the average and standard deviation of the detected frequency, as a function of the observation time H and the ϵ parameter. The plots reveals that the value of the average is not overly affected by these parameters, but the variance clearly is. More specifically, by increasing the ϵ parameter from 0.1 to 0.5 and 0.6, a reduction of the variance is generally achieved because the higher-order frequencies are more easily accredited to the correct frequency value. However, if the ϵ is increased too much, the algorithm

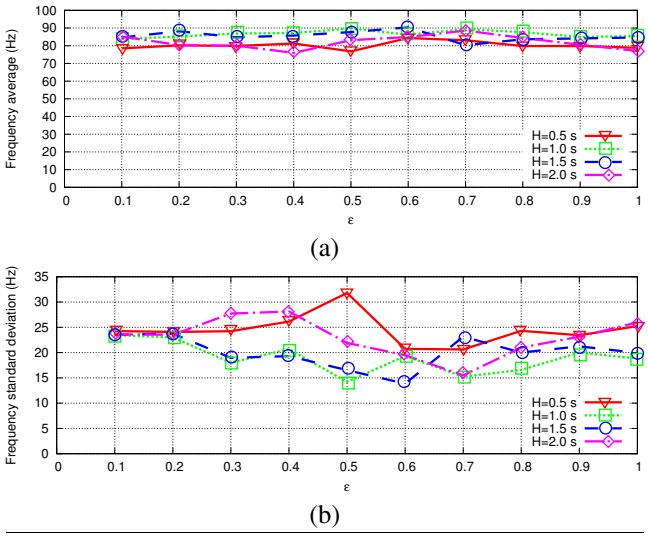


Figure 9. Average (a) and standard deviation (b) of the detected frequency, as a function of the observation time H and the ϵ parameter.

does not distinguish very well between adjacent frequencies and the variance increases.

5.2 Period Detection Precision and Tracing Time

The experimental results shown in this section highlight the precision of the proposed period-detection technique when varying the application tracing time. To this purpose, the `mplayer` multimedia player for Linux has been launched playing a set of mp3 files, which were traced for a different time using our mechanism. At the end of the trace the detected period was recorded. The plot of the amplitude spectrum obtained for different tracing time are reported in Figure 10. In order to enhance readability, values on the Y axis have been normalised to the maximum of 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.

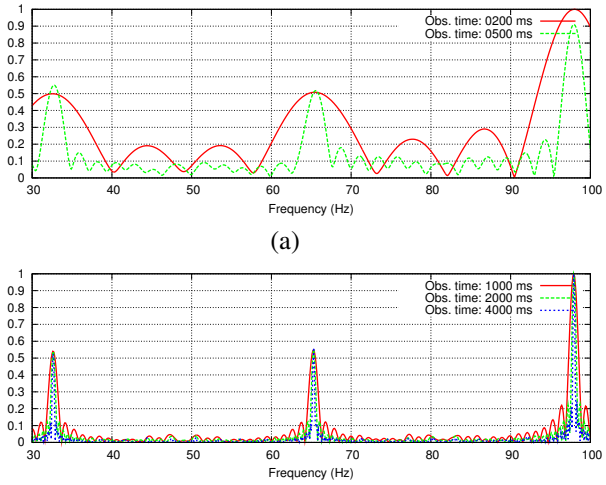


Figure 10. Normalized frequency-transform of the events obtained by tracing `mplayer` at varying tracing time.

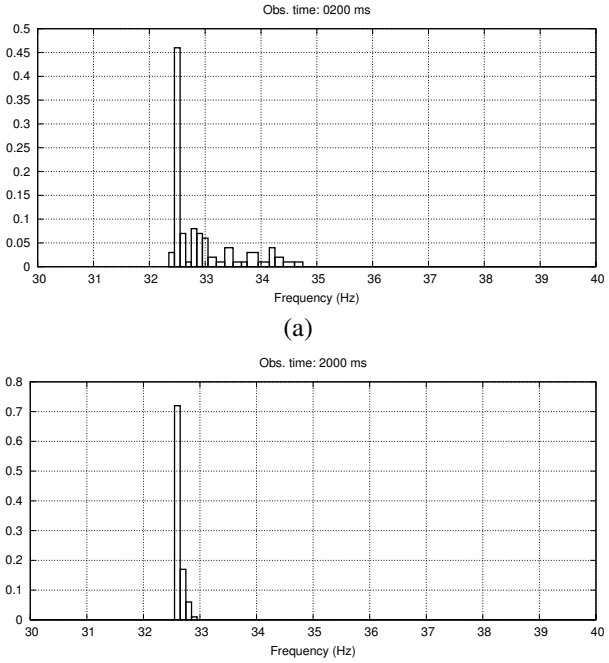


Figure 11. PMF of the frequency detected by LFS++ for `mplayer` at varying tracing times.

Overall load	New reservation	Average freq (Hz)	Std Dev (Hz)	Max (Hz)
0%	-	32.69	6.60	98
15%	(645,4300)	41.67	22.97	97
30%	(1200,8000)	57.98	30.79	95
45%	(1650,11000)	75.03	26.35	92
60%	(2250,15000)	68.47	25.51	93

Table 2. Precision of the period detector with respect to the real-time load in the system. Reservation budgets and periods are in μs, average, standard deviation and maximum values of the detected frequency are in Hz.

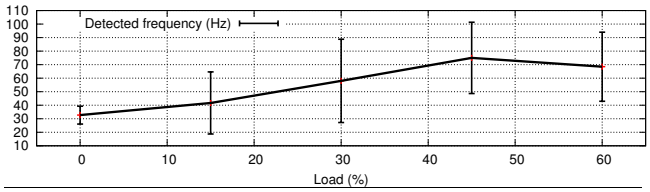


Figure 12. Period detection precision (average frequency +/- standard deviation in error-bars notation), as a function of the background real-time load.

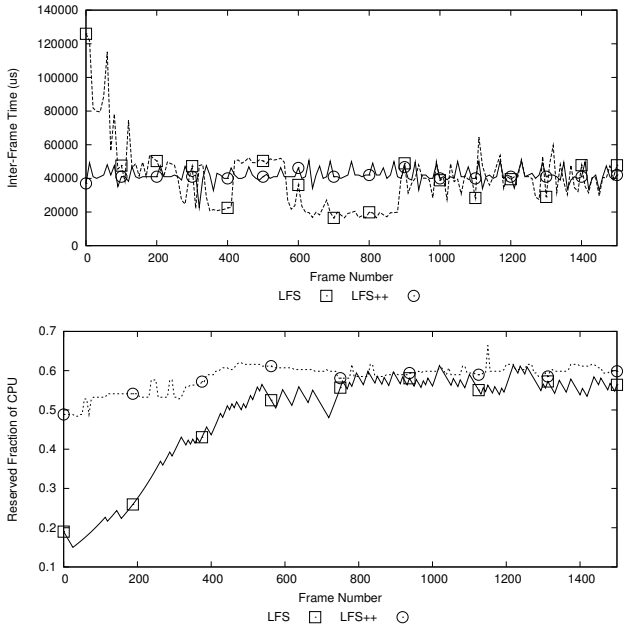


Figure 13. Inter-Frame times and reserved fraction of CPU time for `mplayer` when LFS and LFS++ are used.

5.3 Period Detection Tolerance to Load

In this section, the robustness of the period detection technique is evaluated with respect to interference generated by other real-time applications in the system on the production of the time-stamps by the tracer. A running instance of `mplayer` playing an MP3 song has been traced varying the real-time load in the system. The latter has been synthetically generated by starting instances of a simple real-time periodic application. In Table 2, each row is obtained when adding to the system the real-time application with scheduling parameters in the second column, generating the CPU utilisation reported in the first column. Each run has been repeated 100 times, and the average and standard deviation of the detected frequency value have been measured. The result is reported in Figure 12. As we can see, increasing the background load we also increase the number of time the period detector evaluates a frequency which is an integer multiple of the actual one (at most three times, as seen in the fifth column of the table). Thereby, the average detected frequency (third column) increases with the workload. Also, the reported standard deviation, shown as error bars in the plot and reported as the fourth column in the table, shows the extent of increase in the detected frequency variability as a function of the real-time load on the system.

5.4 Evaluation of the New Feedback Mechanism

The performance improvements achieved by the `LFS++` over the LFS solution [2], have been evaluated in isolation (disabling rate detection, to make the results more reliable) by using `mplayer` as a test application. In particular, we measured the time between the visualisation of two video frames (inter-frame time) and the allocated CPU time.

A large number of experiments (with different videos) have been performed, showing that the new feedback mechanism can adapt the reserved CPU time in a shorter time, and generally produces more stable allocations than `LFS`. An example (corresponding to `mplayer` reproducing a movie with the video at $25 fps$) is shown in Figure 13, showing that LFS is able to control the inter-frame times to less than $80ms$ (the expected inter-frame time is $1000/25 = 40ms$, and an inter-frame time smaller than 80 indicates that the video frame has not been dropped) only after more than 100 frames (4 seconds). This behaviour is easily understandable by looking at the allocated fraction of CPU time, which starts from a low value and grows quite slowly. On the other hand, the new feedback mechanism is able to adapt in a shorter time (almost immediately). Such different behaviours are easily noticeable when looking at the standard deviation of the inter-frame times ($11.287ms$ for LFS and $4.6312ms$ for LFS++), but since the system is underloaded the average values are similar ($39.992ms$ for LFS, and $40.925ms$ for LFS++). This fact can also be appreciated by looking at the Cumulative Distribution Functions (CDFs) of the inter-frame times and reserved fraction of CPU time, which are shown in Figure 14: note that the CDF of the inter-frame times for LFS has a longer tail, and the CDF of the reserved CPU time for the new feedback indicates a smaller variance.

5.5 Complete Feedback Example

After measuring the overhead, tuning the rate detection algorithm, and comparing the new feedback mechanism with the

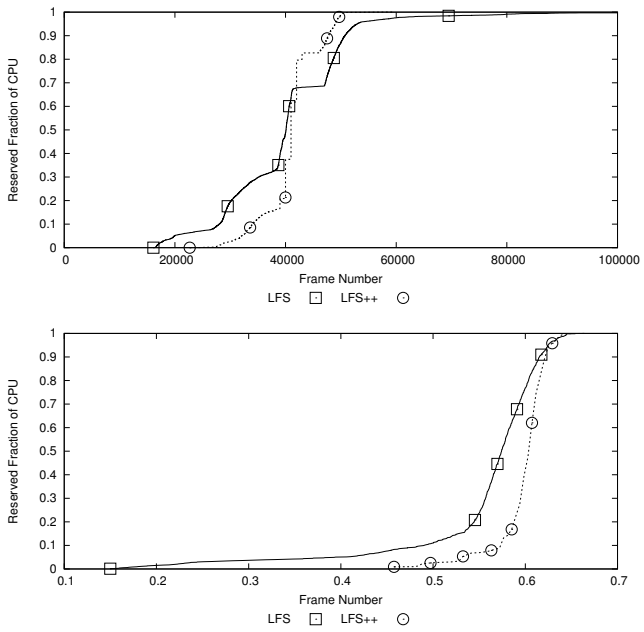


Figure 14. CDFs of the Inter-Frame times and reserved fraction of CPU time for `mplayer` when LFS and LFS++ are used.

Periodic Workload	Average IFT	Std Dev
20%	$40.966ms$	$6.995ms$
30%	$40.934ms$	$7.834ms$
40%	$40.924ms$	$10.943ms$
50%	$40.947ms$	$11.743ms$
60%	$40.959ms$	$16.570ms$
70%	$44.431ms$	$17.865ms$

Table 3. Average values and Standard Deviations for the Inter-Frame times with LFS++ under different real-time workloads.

original one, the proposed LFS++ adaptation has been tested on a real application (`mplayer`) when the system is loaded with some periodic real-time tasks. Table 3 reports the inter-frame times (used as a measure of the perceived QoS) measured when playing a $25fps$ video. The expected inter-frame time is $1000/25 = 40ms$. Note that when the system load increases LFS++ is able to keep the inter-frames time under control (the increasing workload affects the standard deviation, but not the average) until the system is overloaded (with a real-time load of 70%).

6. Future Work and Conclusions

In this paper, we have discussed a framework for scheduling legacy real-time applications in general purpose operating systems. In particular we have identified two mechanisms whose concurrent application promises to disclose important opportunities in scheduling this type of applications. The first mechanism is a frequency domain analyser that uses data collected in the kernel to infer important parameters of the application. The second mechanism is a feedback scheduler that changes the reserved budget to track the computation requirement of the application. Experimental results collected on a prototype implementation of this machinery in the Linux Kernel show how the two technologies combine nicely, overcoming the limitations of previous work by the same authors that simply operated at the scheduler level.

We plan to work on various improvements to the presented mechanism. Concerning the tracer, the current mechanism may be improved on the side of security for usability in a multi-user context. Indeed, the current implementation relies on a single special device which needs to be accessible by the `lfs++` tool, a potentially weak solution from a security standpoint. Another direction can be to trace the transition between blocked and ready (or executing) state in the kernel as an alternative to the system calls. Such information may be collected by enabling tracing options such as `ftrace`, available in recent versions of the Linux kernel, and promises to be more closely related to the task temporal behaviour.

The current LFS++ algorithm for bandwidth adaptation is subject to improvements especially in those situations in which the application undergoes sudden increases of the workload. We envisage the use of a more sophisticated algorithm base a tighter cooperation with the kernel-level scheduler, to cope with this problem. Finally, we plan to investigate on optimal ways to deal with multi-threaded applications and multicore platforms. An interesting possibility is to use a SMP real-time CPU scheduling policy, such as the one presented by some of the authors [7]. 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

[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 L. Palopoli. Legacy real-time applications in a reservation-based system. *IEEE Transactions on Industrial Informatics*, 5(3), August 2009.

[3] 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.

[4] L. Abeni, T. Cucinotta, G. Lipari, L. Marzario, and L. Palopoli. QoS management through adaptive reservations. *Real-Time Systems Journal*, 29(2-3):131–155, March 2005.

[5] G. T. C. Lu, J. Stankovic and S. Son. Feedback control real-time scheduling: Framework, modeling and algorithms. *Special issue of RT Systems Journal on Control-Theoretic Approaches to Real-Time Computing*, 23(1/2), September 2002.

[6] A. Casimiro and P. Verissimo. Using the timely computing base for dependable qos adaptation. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, New Orleans, Louisiana, October 2001.

[7] F. Checconi, T. Cucinotta, D. Faggioli, and G. Lipari. Hierarchical multiprocessor CPU reservations for the linux kernel. In *Proceedings of the 5^{th} International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2009)*, Dublin, Ireland, June 2009.

[8] T. Cucinotta, L. Abeni, L. Palopoli, and F. Checconi. The wizard of os: a heartbeat for legacy multimedia applications. In *Proceedings of the 7^{th} IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia 2009)*, Grenoble, France, October 2009.

[9] A. Easwaran, I. Lee, I. Shin, and O. Sokolsky. Compositional schedulability analysis of hierarchical real-time systems. In *ISORC*, pages 274–281. IEEE Computer Society, 2007.

[10] E. Eide, T. Stack, J. Regehr, and J. Lepreau. Dynamic cpu management for real-time, middleware-based systems. In *Proceedings of the 10^{th} IEEE Real Time Technology and Applications Symposium (RTAS 2004)*, Toronto, Canada, May 2004.

[11] D. Gerhard. Pitch extraction and fundamental frequency: History and current techniques. Technical Report TR-CS 2003-06, University of Regina, Saskatchewan, Canada, 2003.

[12] A. Goel, J. Walpole, and M. Shor. Real-rate scheduling. In *Proceedings of the 10^{th} IEEE Real Time Technology and Applications Symposium (RTAS 2004)*, Toronto, Canada, May 2004.

[13] M. B. Jones, D. L. McCulley, A. Forin, P. J. Leach, D. Rosu, and D. L. Roberts. An overview of the rialto real-time architecture. In *Proceedings of the 7th ACM SIGOPS European Workshop*, Connemara, Ireland, 1996.

[14] C. Krasic, M. Saubhasik, A. Sinha, and A. Goel. Fair and timely scheduling via cooperative polling. In *Proceedings of the EuroSys 2009 Conference*, April 2009.

[15] Y. Krishnamurthy, V. Kachroo, D. A. Karr, C. Rodrigues, J. P. Loyall, R. E. Schantz, and D. C. Schmidt. Integration of QoS-enabled distributed object computing middleware for developing next-generation distributed application. In *LCTES/OM*, pages 230–237, 2001.

[16] J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of the Real Time Systems Symposium*, pages 166–171, 1989.

[17] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7), 1996.

[18] B. Li and K. Nahrstedt. A control theoretical model for quality of service adaptations. In *Proceedings of Sixth International Workshop on Quality of Service*, 1998.

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

[20] P. McLeod and G. Wyvill. A smarter way to find pitch. In *Proceedings of International Computer Music Conference, ICMC*, 2005.

[21] 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.

[22] A. K. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Proceedings of the 7^{th} IEEE Real-Time Technology and Applications Symposium*, pages 75–84, Taipei, Taiwan, May 2001.

[23] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari. AQuoSA — adaptive quality of service architecture. *Software – Practice and Experience*, 39(1):1–31, 2009. ISSN 0038-0644.

[24] 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.

[25] S. Saewong, R. R. Rajkumar, J. P. Lehoczky, and M. H. Klein. Analysis of hierarchical fixed-priority scheduling. In *ECRTS '02: Proceedings of the 14th Euromicro Conference on Real-Time Systems*, page 173, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1665-3.

[26] D. C. 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 Symposium on Operating Systems Design and Implementation*, pages 145–158, New Orleans, Louisiana, USA, February 1999.

[27] C. A. Tsetsekas, S. Maniatis, and I. S. Venieris. Supporting qos for legacy applications. In *ICN*, Lecture Notes in Computer Science, pages 108–116. Springer, 2001.

[28] P. Verissimo and A. Casimiro. The timely computing base model and architecture. *IEEE Transactions on Computers*, 51 (8), August 2002.

[29] T. Yang, T. Liu, E. D. Berger, F. Kaplan, and J. E. B. Moss. 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, December 2008.