The Wizard of OS: a Heartbeat for Legacy Multimedia Applications

Tommaso Cucinotta*, Luca Abeni†, Luigi Palopoli†, Fabio Checconi*
*Scuola Superiore Sant'Anna, Pisa, Italy
Email: {t.cucinotta,f.checconi}@sssup.it
†Università di Trento, Trento, Italy
Email: {luca.abeni,luigi.palopoli}@unitn.it

***Abstract*—Multimedia applications are often characterised by implicit temporal constraints but, in many cases, they are not programmed using any specialised real-time API. These "Legacy applications" have no way to communicate their temporal constraints to the OS kernel, and their quality of service (QoS), being necessarily linked to the temporal behaviour, fails to satisfy acceptable standards. In this paper we propose an innovative way for dealing with these applications, based on the combination of an on-line identification mechanism (which extracts from high-level observations such important parameters as the execution rate) and an adaptive scheduler (specialised for legacy applications) that identifies the correct amount of CPU needed by each application.**

Preliminary experimental results are reported, proving the effectiveness of the proposed idea in providing a widely used multimedia player on Linux with appropriate QoS guarantees, through an appropriate choice of the scheduling parameters. Finally, a detailed road-map is presented with the possible extensions to the approach.

I. Introduction

In recent times, general-purpose (GP) computers have emerged as one of the most effective means to produce, store and distribute multimedia contents. Very frequently personal computers operated by general-purpose Operating Systems (OS) are used for video and audio streaming, for editing home-made movies, for video conferencing. From the perspective of the OS, such applications are very challenging. They are time-sensitive in that the Quality of Service (QoS) provided by the application to the user depends on the respect of some temporal constraints. On the other hand, such timing requirements are not hard, in fact moderate and occasional delays are acceptable as long as the anomaly is kept in check.

Since applications usually share a common computing platform, a prominent issue is the development of scheduling policies that can be used to ensure their correct and timely evolution. A very interesting technology is the one of soft real-time schedulers, and specifically the *resource reservations* [1]. These algorithms ensure a correct temporal partitioning of the system resources whereby each application is guaranteed a share of its computing power regardless of the behaviour of the other applications present in the system. This solution has been complemented by adaptive mechanisms to figure out the CPU requirement of time varying or unknown applications and choose the scheduling parameters appropriately [2], [3]. However, an assumption invariably made by such algorithms is that the application is structured as a (typically periodic) stream of jobs and that it makes use of some specialised API available on the underlying OS for: 1) communicating the required scheduling parameters, 2) notifying the start and the termination of each job. This way it is possible for the system to sample the value of some quantities related to the QoS of the application and take corrective control actions (by changing the bandwidth allocated to the task) as needed.

Unfortunately, the programming interface available today on a GPOS for real-time computing is mostly limited to the POSIX real-time extensions [4]. This API provides fixed priority scheduling, timers and mechanisms for bounding the *priority inversion* problem. While such features are very useful for embedded applications, they do not prove so effective for GP multimedia applications, for which the availability of a soft real-time scheduler is much more important.

Only recently have real-time APIs supporting this class of time-sensitive applications been proposed, as a result of several research projects. This is for example the case of the architecture developed in the context of the FRESCOR European Project[1], or the real-time services which are being designed in the context of the IRMOS European Project[2].

While open-source applications may be modified in order to take advantage of the new real-time OS functionalities, for *legacy applications*, the source-code is not usually available, and the constraints on their life-cycles (subject to commercial policies) hinder the evolution of the application. For this class of applications, designers contrive to ensure an acceptable timing behaviour by heuristic and often ineffective solutions, such as the generous use of internal buffering (which introduces latency and decreases the interactivity level of the application). The main problem with legacy applications is that they do not communicate to the OS the start-time and end-time of their jobs [5]. Therefore, there is no way for the system to associate

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".

[1]More information is available at http://www.fresocr.org.
[2]More information is available at http://www.irmosproject.eu.

deadlines to jobs.

The purpose of this work is then to extend the benefits of real-time scheduling to this kind of applications, without imposing any modification to the applications themselves. This is a challenging and multifaceted problem whose solution requires: 1) the ability to correctly infer such important parameters as the activation period of the application, 2) an adaptation of the scheduling parameters to the application ensuring its correct and timely progress. We address the first problem by a combination of two technologies: a tracer, that extracts a time-series of events from the kernel, and a frequency-domain analyser, that extracts the most important frequencies from the time series and identifies the fundamental (pitch) frequency to guess the execution rate of the application. We address the second problem using a feedback scheduler (initially presented in [5]) that, observing the evolution of some scheduling parameters, identifies the computation requirements of the task and adjusts the reserved bandwidth accordingly. One of the crucial points made in this paper is that the effectiveness of the feedback scheduler is greatly magnified by the availability of the task period, reconstructed by the frequency analyser.

A. Paper Structure

The paper is organised as follows. In Section III, the problem of the identification of optimum scheduling parameters, and specifically of the period, for legacy multimedia applications is introduced. In Section IV, the general proposed methodology for addressing the problem is presented, while in Section V it is validated by practical experiments conducted on a prototype implementation of the proposed technique on Linux. In Section II, the related work in the research literature is briefly overviewed, and in Section VI the road-map for further research on the topic is presented. Finally, conclusions are drawn in Section VII.

II. Related Work

The problem of dynamically adapting the amount of CPU time reserved to an application (introduced in Section I) can be addressed by applying feedback control to real-time scheduling [6], [7], [2], [8], [3]: while the applications execute, their real-time behaviour is monitored and corrective actions are taken changing the scheduling parameters so that specified QoS objectives are met. If additional assumptions can be made on the application, it is possible to use application level feedback such as the one shown in [9]. However, both the used of a specialised API and (all the more) the availability of application level adaptive mechanisms cannot be assumed in the context of legacy applications.

The problem of finding an appropriate allocation for legacy applications is known in the Internet community. In particular in [10] the authors propose an architecture using proxy servers to determine the network requirements of Internet applications.

In the domain of real-time scheduling, there has been some work in dynamically inferring task parameters for legacy applications. For example, BEST [11] tries to infer the task periods by monitoring the times at which the tasks enter the scheduler ready queue. Compared to BEST, the approach presented in this paper separates the scheduling algorithm from the task parameters estimation (allowing to easily combine different reservation-based scheduler with different adaptation mechanisms and period detection heuristics), and uses a more advanced algorithm for detecting periodic tasks.

Other techniques that could possibly be used as a basis for adaptive scheduling of legacy applications have been proposed in the past [12], [13], but to the best knowledge of the authors the first technique developed explicitly to this purpose is in [5], where a feedback scheme for legacy applications was proposed (by almost the same authors of this paper) that uses a simple multiplicative/additive scheme to identify the resource requirements by using a coarsely quantised feedback variable.

In this paper, the latter approach is enhanced and made more effective by complementing the feedback controller with a trace analyser that extracts meaningful information on the task (in our case the period of the tasks) from the time-series of events recorded in the Kernel. This analyser requires the use of two distinct technologies: a tracer component inside the kernel and a spectrum analyser to identify the period of the task. The latter problem is well known in the literature of digital processing of sound signals, where different approaches have been developed to extract the pitch and identify the fundamental frequency [14], [15]. Such approaches served as a good starting point for our analyser, but we had to adapt them to the analysis of a time-series of events.

As far as the problem of tracing events in the kernel is concerned, there are various mechanisms available, like Linux Tracer Toolkit (LTT/LTTng)[3], or the more recent `ftrace`[4] tracer integrated into the mainstream kernel.

III. Problem Presentation

To provide legacy applications with a resource allocation as tight as possible to their actual requirements, the periods of such applications must be correctly identified. The need for such a period identification mechanism is shown in this section.

After the introduction of background concepts and definitions in Section III-A, the investigation is carried on from the theoretical real-time scheduling perspective in Section III-B. Then, a set of experimental results are shown in Section III-C, which, confirming the theoretical expectations, constitute a fundamental motivation of the presented work.

A. Background and Definitions

In real-time theory, a system is often modelled as a set $\Gamma = \{\tau_i\}$ of real-time tasks; in this paper, 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 very simple yet popular model of a real-time task is the one where a task τ_i is modelled as a stream of *jobs* and is described by a pair (C_i, P_i): C_i is the worst-case execution

[3]More information is available at http://ltt.polymtl.ca.

[4]More information is available at: http://lxr.linux.no/linux+v2.6.30/Documentation/trace/ftrace.txt.

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, and this represents an implicit temporal constraint.

In this work, a legacy application τ_i (either a single task or a set of tasks) is guaranteed by using a *resource reservation*, which allows to reserve to τ_i an amount of time Q_i^s every period T_i^s. The reservation allows to control both the execution rate of the application (the allocated fraction of the CPU is Q_i^s/T_i^s) and its responsiveness (the reservation period T_i^s controls the allocation granularity).

The scheduling algorithm used in this work to implement the reservation behaviour is the Constant Bandwidth Server (CBS) [16], which implements CPU reservations based on EDF. 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. More formally, the CBS works by maintaining two variables for every reservation: the server budget q_i (used for accounting) and the current scheduling deadline d_i^s (used for assigning a priority to the scheduled task and for enforcement). Such variables are updated as follows:

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

Summing up, when scheduling a legacy application through a CBS (Q_i^s, T_i^s), the problem is to infer reasonable values for Q_i^s and T_i^s that allow to serve the application so that it can meet its timing constraints.

B. Period and Budget Adaptation

If the WCET C_i and the period P_i of a real-time task τ_i are known, the traditional approach to reservation-based scheduling exploits such knowledge to set $T_i^s = P_i$ and $Q_i^s = C_i$ so that all the task's deadlines are met [16].

However, for a legacy real-time application, the enclosing reservation providing scheduling guarantees may not necessarily know the exact period. Therefore, identifying a correct budget allocation that allows the task to respect its deadline is not an obvious problem.

If a reservation (Q_i^s, T_i^s) is used to serve a single task τ_i, it is possible to investigate the relationship between (C_i, P_i), (Q_i^s, T_i^s), and the QoS provided to the task (in terms of missed deadlines). This analysis can be performed by using the concept of *supply-bound function* $Z_{Q_i^s, T_i^s}(\cdot)$, describing the worst-case amount of time provided by a (Q_i^s, T_i^s) reservation to a task τ_i starting at time 0.

Since the reservation abstraction guarantees that Q_i^s units of CPU time are provided to τ_i in a reservation period T_i^s, the worst-case CPU allocation corresponds with the case in which the task is scheduled at the end of the reservation period. Hence, $Z_{Q_i^s, T_i^s}(t)$ is 0 if $t < T_i^s - Q_i^s$, increases with a gradient 1 if $T_i^s - Q_i^s \leq t < T_i^s$, is equal to Q_i^s if $T_i^s \leq t < 2T_i^s - Q_i^s$, etc... Some examples of various supply-bound functions with equal utilisation (Q_i^s/T_i^s) but different server periods are shown in Figure 1, and the resulting supply-bound function is:

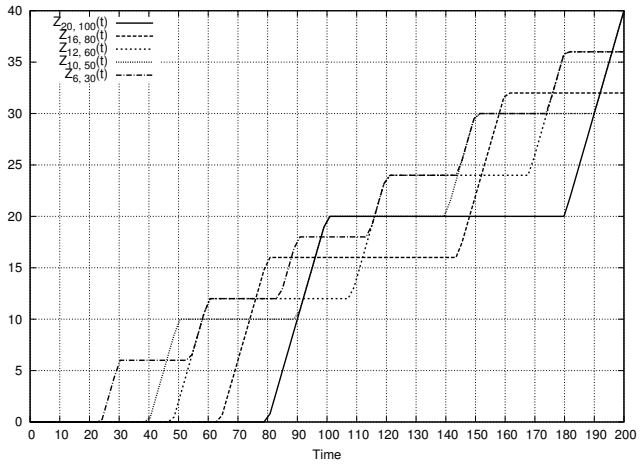

Figure 1. Supply-bound function for different periods, at equal utilisation of the resource.

$$Z_{Q_i^s, T_i^s}(t) = \begin{cases} hQ_i^s & \text{if } t \in]hT_i^s,\ (h+1)T_i^s - Q_i^s] \\ t - (h+1)(T_i^s - Q_i^s) & \text{otherwise} \end{cases}, \quad (1)$$

with $h \triangleq \left\lfloor \frac{t}{T_i^s} \right\rfloor$. Note that a similar concept of supply-bound function is often used in hierarchical scheduling analysis - for example, see [17]. The difference between the function presented here and the one used for analysing hierarchical scheduling systems is that the scenario under investigation is much simpler due to the fact that a single task is being enclosed within the reservation.

Based on this definition, one possible test that guarantees that every deadline is respected is based on *Time Demand Analysis* [18]. Such a test checks that the amount of time provided by the reservation to the task before the deadline is enough to serve a job (i.e., $\geq C_i$). In other words, a time-instant exists such that the supply-bound function for the reservation is greater than the worst case execution time of the real-time task:

$$\exists t \in [0,\ P_i] \ \text{ s.t. } C_i \leq Z_{Q_i^s, T_i^s}(t) \quad (2)$$

Such test may be used to compute the minimum budgets that should be granted to the reservation in order to allow the served real-time task to meet all of its deadlines. Figure 2 (a) reports the minimum values of Q_i^s needed to correctly schedule a real-time task with a period of $P = 100ms$ and a WCET of $C = 20ms$ (i.e., a utilisation of 20%). Figure 2 (b) reports the corresponding fraction of CPU Q_i^s/T_i^s reserved to the task. As the plots reveal, setting a completely wrong reservation period (and under the assumption that the feedback-based

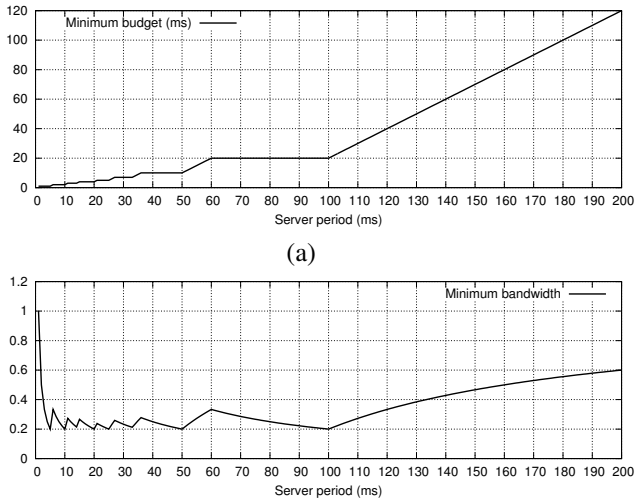

Figure 2. Minimum value of Q_i^s (a) and corresponding fraction of CPU bandwidth Q_i^s/T_i^s (b) required to correctly schedule a $(20ms, 100ms)$ real-time task.

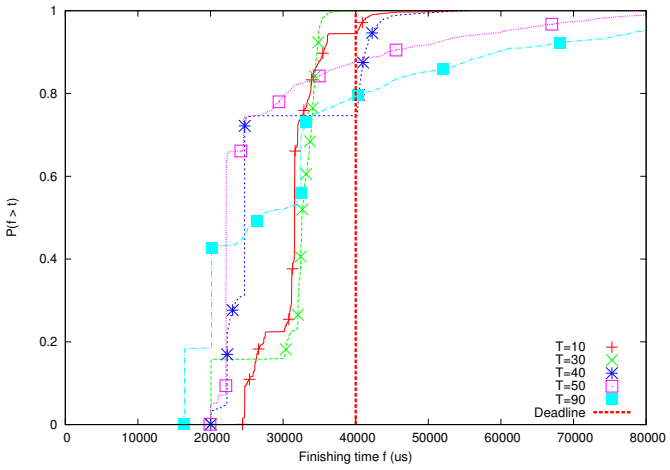

Figure 3. CDF of the response times for a task with period $P = 40ms$ and various reservation periods T^s.

scheduler manages to identify the corresponding minimum budget), may lead to waste of CPU bandwidth. Indeed, even if we rule out the choice of very small periods leading to an unrealistic overhead, it is possible to increase of more than a half the bandwidth requirements of the task, as compared to the actual requirements. If the server period is *greater* than the task period, then the situation goes even worse, because the bandwidth waste grows uncontrolled. On the other hand, the picture also shows that the best budget assignment is found in correspondence of a server period equal to the actual task period, or an integer sub-multiple of it. However, the choice equal to the task period is the most robust, because small errors in terms of the task period determination are quite well tolerated and lead to the lowest bandwidth wastes. This is also highlighted from Figure 1, showing that, among the supply functions with a utilisation equal to the one of the served task (20%), only the ones with a server period equal to the task period (100) or an integer sub-multiple preserve a supply value of 20 time-units in correspondence of the task deadline (100), while the other curves exhibit lower values.

In the previous example, a single task is being considered, but generally a real-time application may be composed of multiple threads of execution with different real-time parameters. When using a single reservation for serving all those threads, the analysis presented above can be extended by reusing concepts from the theory of hierarchical real-time systems [19], [20], [21], [17]. In particular, this requires to use a different definition for $Z_{Q_i^s, T_i^s}(t)$ and to consider a *demand-bound function* instead of the WCET in Equation 2.

C. An Example

The theoretical analysis presented in the previous subsection is confirmed by some simple examples with two real-time applications (each of them composed by one periodic task) executed on real hardware. To this purpose, each real-time application has been assigned a reservation with arbitrarily set server periods, while the budget was dynamically computed by the LFS algorithm [5] (see Section IV-A) to reduce the number of missed deadlines.

Figure 3 shows the Cumulative Distribution Function (CDF) of the response-time of one of the periodic real-time tasks (having period $P = 40ms$), when it is controlled by LFS, under various choices of server period. The figure shows that picking a server period smaller than or equal to the application period attains a quite good performance. In fact, the CDFs for $T^s < P$ show very short tails after $40ms$ (a minimum amount of deadline misses is inherent to the way LFS works). The CDFs for $T^s > 40ms$ are qualitatively similar to the plots corresponding to $T^s = 50$ or $T^s = 90$, and have been removed from the figure to make it more readable. However, looking at Figure 4 (which shows 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 (again, some curves for $T^s > 40ms$ are not shown for the purpose of clarity, as they are similar to the $T^s = 40ms$ curve). On the other hand, the two figures show that using any other value as server period either leads to a significant waste of bandwidth, or to poor performance in terms of deadline misses.

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 III-B. This was also expected, because the discussion in Section III-B refers to the minimum theoretical budget needed to schedule the real-time task hosted by the reservation, and it does not deal with such issues as how such budget may possibly be found. Due to the particular way LFS works, as it will become more clear in Section IV-A, the necessary budget is greatly over-estimated, in such cases.

The experiments above show that the best results, both in terms of application performance, and of bandwidth allocated within the system, are achieved when the server period is set

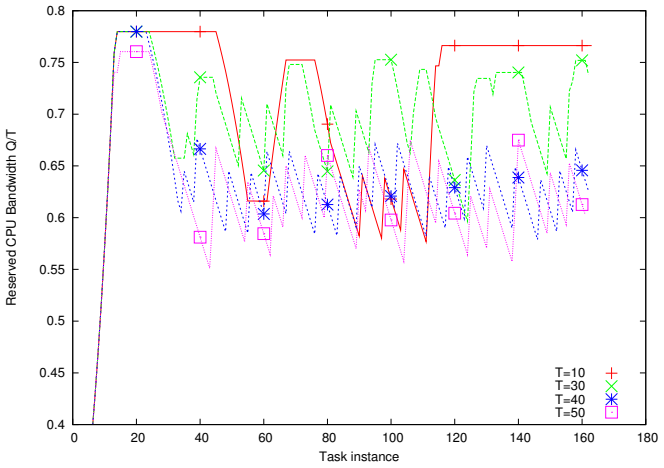

Figure 4. Fraction of CPU bandwidth allocated to a periodic task $(C = 20ms, P = 40ms)$ by LFS.

equal, or as close as possible, to the task period. This is why in this paper the problem of period detection for legacy real-time applications is investigated.

IV. Proposed Approach

The approach proposed in this paper is summarised in Figure 5. A real-time application is monitored while it is running, by intercepting the events which are needed (or useful) for inferring its period. Then, the times at which these events have been generated are analysed to infer the application period, if any. Then, a first rough estimate of the budget needed by the application (with the estimated period) is built, and a CPU reservation is attached to the application threads (by scheduling them through a CBS). Then, the maximum budget Q_i^s is continuously adapted on-line while the application is running, by using the Legacy Feedback Scheduling mechanism. The period estimation process is repeated periodically to gather updated information on the application period. As shown later, this process adds a little overhead to the system which is perfectly sustainable.

The events that are considered as mostly relevant for the purpose of period identification are the ones corresponding to when the application blocks waiting for either a signal or the arrival of a packet from the network or disk, and when it wakes up later. Such events usually occur in correspondence of the call of some blocking system call, like `read()`, `usleep()`, `nanosleep()`, etc.

Therefore, in this preliminary work, a tracing program named `qostrace`[5] has been developed making use of the standard `ptrace()` call available on Linux to intercept the system calls made by an application at run-time. This program can be used to trace an application while it is running: the application is suspended in correspondence of each system call entry and exit, and control is passed to the tracer process which can perform various kind of inspections on the traced application before it continues its execution. For the purpose of this paper, only the time at which system calls were entered and left was relevant, so the traced program was suspended only for the minimum necessary time (see Section V-B for overhead measurements).

[5]For the reader convenience, the program is available at the URL: http://retis.sssup.it/~tommaso/eng/papers-estimedia09.html.

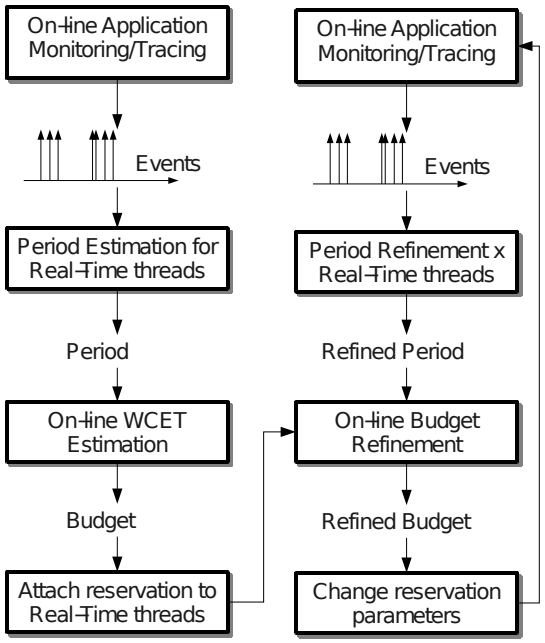

Figure 5. Scheme of the proposed approach.

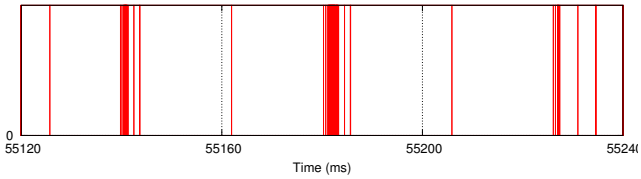

Figure 6. Events generated by `mplayer`.

Figure 6 shows an excerpt of the set of events generated by the `mplayer` software while playing a video at $25fps$, i.e., with a period of $40ms$. As the picture highlights, every application period there is a conspicuous number of events in correspondence of the activation of each application job.

In the proposed approach, the sequence of time instants $(t_1, \ldots, t_N)$ at which these events occurred, over a sufficiently long time-window, is reinterpreted as a time-continuous signal $f(\cdot)$ with a null value everywhere, except at the times in which events were detected, where it exhibits Dirac's Deltas:

$$f(t) = \sum_{i=1}^{N} \delta(t - t_i). \tag{3}$$

Then, the first harmonic of this signal is taken as the application period.

As an example, Figure 7 plots the frequency-transform obtained for the events collected from the `mplayer` run whose excerpt is shown in Figure 6.

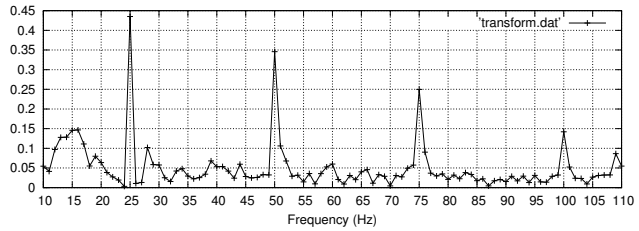

Figure 7. Frequency-transform of the events generated by `mplayer`.

In order to compute the first harmonic, the following heuristic algorithm is proposed in this preliminary work:

1) compute a sampling of the modulus of the frequency-transform of $f(\cdot)$ over a target frequency-range $[f_{min}, f_{max}]$, at steps of δf, by means of the following formula:
$$\mathcal{F}(f) = \left| \sum_{i=1}^{N} e^{-j2\pi f t_i} \right|; \quad (4)$$
2) peaks (local maximum values) of the obtained sampled frequency-transform are identified (ordered from the smallest to the greatest frequency value): $f_1, \ldots, f_m$;
3) only peaks for which $\mathcal{F}(\cdot)$ is higher than K times its average value $\overline{F}$ are considered as candidate frequency values;
4) if the resulting set of candidate values is empty, then declare the application as non-periodic and **terminate**;
5) if the resulting set of candidate values is composed of at most $M \geq 1$ values, then simply pick the frequency f_i corresponding to the maximum of $\mathcal{F}(\cdot)$, and **terminate**;
6) a *weighted linear-regression* is performed among the candidate frequencies f_i, using the $\mathcal{F}(f_i)$ values as weight factors, resulting into a regression line: $f_i = F_1 i + F_0$;
7) if the squared error of the linear-regression is smaller than a threshold E, then the frequency peak f_i which is closest to the regression line F_1 coefficient is picked as the detected frequency, and **terminate**;
8) otherwise, simply the frequency f_i corresponding to the maximum of $\mathcal{F}(\cdot)$, among the candidate frequencies.

A much simpler algorithm may be simply obtained by directly picking the maximum of the $\mathcal{F}(\cdot)$ frequency-transform (which presumably corresponds to the first peak). However, sometimes it may happen that the maximum is not found in correspondence of the first harmonic, but of a larger harmonic. However, for the class of multimedia applications that have been traced (video and audio decoders and players), usually it happens that most (but rarely all) of the candidate frequencies identified at step 3 are non-first harmonic, i.e., all integer multiples of the same value. The purpose of the linear-regression is to identify the first harmonic also in these cases.

Note that E, M, f_{min}, f_{max}, δf and K constitute tunable parameters of the algorithm. In what follows, $E = 0.1$, $f_{min} = 10Hz$, $f_{max} = 200Hz$, $\delta f = 1Hz$, $K = 2.5$, and $M = 2$ have been used.

A. Legacy feedback algorithm

To properly serve a time-sensitive task (or set of tasks), the two reservation's parameters Q_i^s and T_i^s have to be computed. While the reservation period T_i^s can be selected by using the techniques presented above, the maximum budget Q_i^s can be adapted through *feedback scheduling*. For example, the execution time of the tasks can be monitored, and Q_i^s can be assigned based on the monitored values, or the Legacy Feedback Scheduler (LFS) [5] can be used[6].

LFS applies feedback scheduling to "unaltered" legacy applications by defining a scheduling error $\epsilon_i = d_i^s - t$ (instead of $\epsilon_i = d_i^s - d_i$, as in adaptive reservations) and by using such scheduling error ϵ_i to adapt Q_i^s: if $\epsilon_i > T_i^s$ then we can deduct that the application has not been given enough time and Q_i^s should be increased. A more formal definition of the LFS algorithm follows:

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

Various feedback functions $f()$ have been proposed; in this paper, the simplest one, LFSg, (which is more robust against uncertainties in the tasks periods) is used:

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

V. Experimental Results

In the framework proposed above, we have implemented two blocks: the estimator of the activation rate of the threads, and the online mechanism for period adaptation. In this section we show the results we achieved on a real-life multimedia application. In particular, we will stress on the effectiveness of the period analyser and on its efficiency (the overhead it introduces). Then we will show the improvement in the efficiency of the feedback scheduler when it uses the parameters produced by the period analyser.

A. Observation Period and Precision

First, the precision of the proposed technique, in relation to the duration of the observation time-period, is analysed. To this purpose, the `mplayer` multimedia player for Linux has been launched multiple times on the same movie, and the proposed tracer was attached at approximately the same relative time-instant from the start of the play, but at varying durations of the observation. Figure 8 reports the obtained frequency-transforms in the various cases of observation durations ranging from 0.2 seconds to 4 seconds. It is clear that, increasing the observation duration, the peaks of the frequency-transform corresponding to the real application frequency

[6]Notice that Adaptive Reservations cannot be directly used because legacy applications do not use a real-time API.

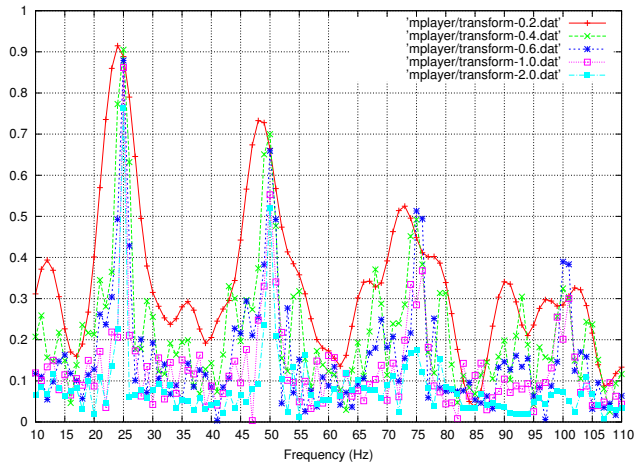

Figure 8. Frequency-transform of the events obtained by tracing `mplayer` at varying tracing durations.

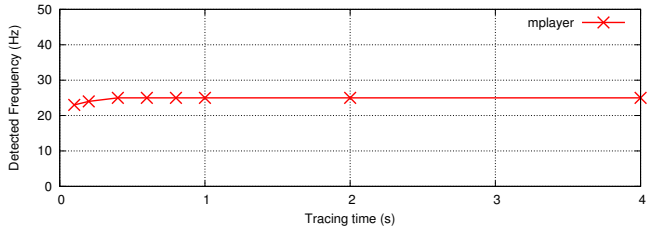

Figure 9. Period automatically detected over the frequency transforms of Figure 8, at varying tracing durations.

$(25Hz)$ become much more neat. Moreover, for an observation period of 0.2 seconds or below, the first harmonic is detected almost correctly, with a little error. On the other hand, with observation durations from 0.4 seconds (and beyond), the application frequency is detected without mistakes, but the frequency peak becomes more evident and sharp by increasing the observation duration.

Finally, Figure 9 shows the period as automatically detected by the detection algorithm heuristic. As it can be seen, with very short tracing times like $0.2s$, corresponding to barely 5 jobs of the player, the frequency is slightly underestimated, while increasing the tracing time to $0.4s$ (corresponding to barely 10 jobs) allows to properly detect the working frequency. Increasing the tracing time beyond $0.4s$ does not seem to lead to any advantages.

Summarising, the preliminary experimental validation conducted over the `mplayer` application shows that a tracing time of barely 10 jobs is sufficient for detecting the period of the application with a sufficient precision. This confirms the usability of the methodology sketched out in Figure IV, in which the application is periodically traced in order to detect possible variations in its run-time period. However, in order to verify the feasibility of the approach, it is very important to gather overhead measurements of the introduced mechanism, what is done in the section that follows.

Table I
MEASURED OVERHEAD, OBTAINED WITH DIFFERENT TRACERS.

Tracer in use	Total duration (s)	System time (s)
None	21.02 ± 0.14	0.15 ± 0.02
`strace`	22.03 ± 0.11	0.84 ± 0.04
`qostrace`	21.60 ± 0.16	0.51 ± 0.04

B. Tracer Overhead

Some experiments have been performed to gather information about the run-time overhead caused by the proposed rate estimation mechanism on the (legacy) real-time applications that are being traced, as well as on the system. In this section, the focus is entirely on the tracing mechanism which is needed by the period detection overhead (in other words, the run-time overhead due to budget adaptation is not measured. For the LFS algorithms, such details can be found in the original paper [5]).

The simplest way to collect the data needed for the rate estimation algorithm is to use the `strace` program, which is available on Linux (and on other Unix-like systems) and provides the needed tracing functionality. The `strace` program works by using the `ptrace()` system call, and risks to exhibit a considerable overhead due to the behaviour of the program (as it works by intercepting all the system calls and important events (signals, etc...) and by writing a well-formatted and human-readable report on the standard error while tracing the program). Obviously, the overhead caused by `strace` depends on the amount of system calls (or traced events) generated by the traced program.

To reduce the tracing overhead, a custom tracer has been developed, exploiting the same principle as `strace`, named `qostrace`. This program, given the `pid` of the process to trace on the command-line, attaches to it by means of the `ptrace()` system call. However, it limits itself to store into an in-memory array the set of events of interest, along with their time of occurrence, then it performs on the data set the required computations.

A first rough measure of the overhead has been performed through the use of the `time` utility while transcoding a video with `ffmpeg`[7]. First, the program is run without any tracer active, then the program is traced with `strace` and finally with `qostrace`. The average duration of the transcoding process and the time spent inside the kernel are shown in Table I; all the values are in seconds, the averages are taken over ten repetitions, the standard deviation for each measure is shown. In this case the overhead imposed by `strace` is about the 4.8%, while with `qostrace` it goes down to the 2.8%.

Note that the overhead is quite high because the traced program is frequently invoking system calls. The tracing overhead has also been measured on a second program, and precisely a video player based on `ffmpeg` and GDK/GTK[8]. Such video

[7]More information is available at http://www.ffmpeg.org.
[8]More information is available at http://www.gtk.org.

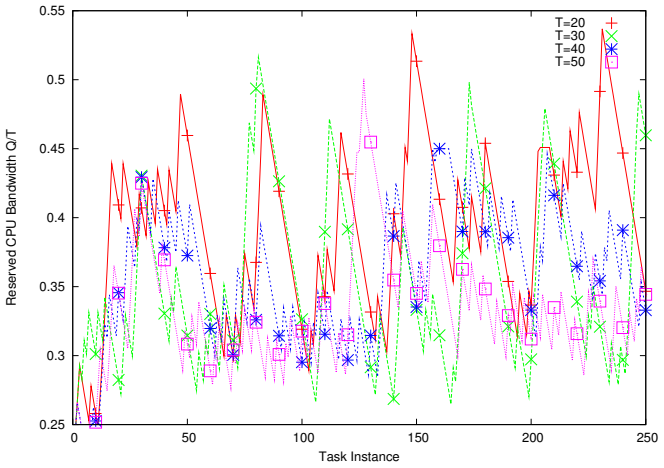

Figure 10. CPU bandwidth allocated to the player as a function of time.

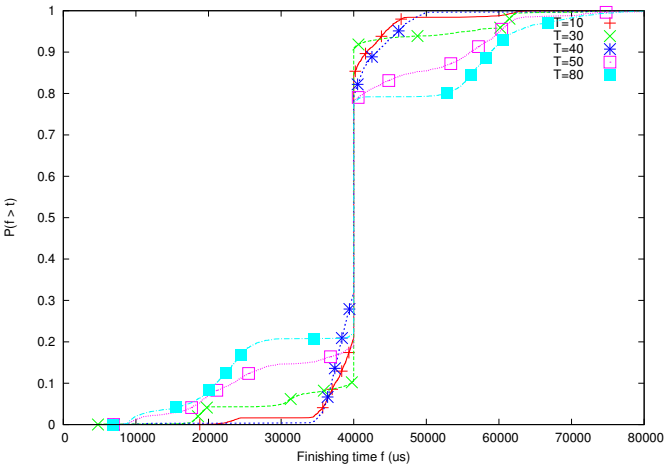

Figure 11. CDF of the inter-frame times for the player.

player is based on a periodic task which periodically reads a video frame from a stream by using `libavformat`, decodes it by using `libavcodec`, converts it to the RGB colour-space, and finally displays the RGB data by using GDK. The time needed to read, decode, convert to RGB, and display a video frame (corresponding to the execution time of a real-time job) has been measured when the program is traced and when it is not traced. Note that this video player spends most of the time in decoding a frame and converting it to RGB (without invoking any system call), and the number of invoked system calls is quite low. As a result, the tracing overhead is lower than the one measured in the previous experiment.

The experiment has been repeated 30 times, computing the averages and the 95% confidence intervals of the execution times. When the video player is not traced, the average time needed to read, decode, convert, and display a frame is $22.313ms$, and the 95% confidence interval is $0.073ms$. When the program is traced with `strace`, the average time is $22.965ms$, which is 2.9% higher, with a 95% confidence interval of $0.137ms$. When `qostrace` is used instead, the average time is 22.559, with a 95% confidence interval of $0.085ms$; as a result, the increase in the execution times is about 1.1%.

C. Efficiency in Resource Allocation and Period

Some experiments show the relationships between the reservation period T_i^s (set equal to the estimated task period P_i), the accuracy of the CPU allocation performed by LFS, and the QoS achieved by the application. Such experiments have been performed by using an implementation of the CBS in the Linux kernel [22] and playing an MPEG4 stream at $25fps$ (hence, the player has a period equal to $P_i = 1000/25 = 40ms$) and using LFS to dynamically adapt the amount of CPU time reserved to the player task. The experiments showed that if $T^{sample} >> T_i^s$ then the exact T^{sample} value does not significantly affect the performance, and the reported experiments have been ran with $T^{sample} = 500ms$. The inter-frame times (intervals between the visualisations of two consecutive frames) and the allocated CPU bandwidth Q_i^s/T_i^s have been measured when LFS uses different reservation periods T_i^s.

Table II
INTER-FRAME TIMES AND ALLOCATED CPU AS A FUNCTION OF T_i^s.

T_i^s (ms)	90 and 95 percentile of the inter-frame times (ms)	average / maximum allocated CPU bandwidth
20	41.803 / 44.383	0.385653 / 0.536840
30	40.1 / 55.639	0.358286 / 0.516393
40	43.189 / 46.109	0.325452 / 0.452615
50	56.074 / 60.148	0.336614 / 0.500588
60	59.066 / 62.393	0.344995 / 0.492892
70	53.076 / 61.370	0.348372 / 0.486546
80	58.853 / 62.339	0.342492 / 0.449763
90	60.319 / 68.516	0.324594 / 0.404568
100	53.959 / 74.186	0.318421 / 0.421542

The fraction of CPU time reserved by LFS to the player during the first 250 jobs (to make the figure more understandable, only a small number of jobs have been displayed) is shown in Figure 10. Note that the peak CPU bandwidth allocated if $T_i^s = 20ms$ or $T_i^s = 30ms$ is more than 50%, whereas the peak CPU bandwidth allocated for $T_i^s = 40ms$ is less than 45%. Hence, if $T_i^s < 40ms$ LFS tends to be more aggressive, causing some transient CPU over-allocations. Since the amount of time allocated to the player if $T_i^s > P_i$ is quite similar to the $T_i^s = P_i = 40ms$ case, all the plots for $T_i^s > 50ms$ have been removed from the figure (to make it more readable). Notice that this figure shows that $T_i^s < 40ms$ causes an overestimation of the allocated bandwidth, but does not clearly show the problems with $T_i^s > 40ms$. Such problems are visible when looking at the player's performance: in fact, if $T_i^s > 40ms$ the inter-frame times tend to increase, as shown Figure 11 which displays the CDF of the inter-frame times (again, some curves have been removed from the figure to make it more readable). From the figure, it is easy to see that the case with $T_i^s = 40ms$ is the one performing better (the tail of the CDF is shorter - in the ideal case, the CDF should be a step going from 0 to 1 at $40ms$). Table II reports

the 90-percentile and 95-percentile of the inter-frame times, and the average and maximum allocated bandwidth for more values of the server period, confirming that $T_i^s = 40ms$ is the best choice.

VI. Future Work

The authors plan to work on various improvements to the mechanism presented in this paper, on different aspects: the tracing mechanism, the period detection algorithm, the legacy feedback-based controller, and the support for multi-thread applications. A detailed description of the road-map follows.

A. Tracing mechanism

The tracing program used in this paper, `qostrace`, uses the `ptrace()` system call, implying the need for suspending the traced process at each occurrence of a relevant event, passing control to the tracing program, then continuing. Even if the overhead incurred by such a mechanism, as measured in the previous section, is sustainable for a large class of systems, it is useful to search for mechanisms which may possibly have a lower impact on the applications that are running.

Therefore, it is foreseen to investigate, in the future, on the use of a kernel-level tracing mechanism registering scheduling events of interest for the traced application, for example the time instants at which the traced process blocks and unblocks. One possibility that we are evaluating is the one of recurring to the use of low-level tracers that already exist for the Linux kernel, like the Linux Trace Toolkit Next Generation (LTTng)[23], [24], the `utrace` [25] and `uprobes` framework [26], or the `ftrace` [24] tracer recently integrated into the mainstream Linux kernel. Such tools can provide a plethora of information on the timing behaviour of the kernel and running applications. However, it must be considered that they are developed and maintained mainly as debugging helpers, and are not designed to be actually enabled in a "production" environment. For example, the `sched_switch` tracer of `ftrace`, when configured into the kernel and enabled at runtime, exports by means of the `debugfs` information about all the scheduling events that occur into the system, not only of the traced processes, therefore it is expected that the implied overhead be much higher than strictly necessary. Moreover, `ftrace` requires administrator's privileges for being used.

Therefore, while the above mentioned tools (as well as the `ptrace()` system call used in this paper) may be leveraged to build prototypes and proof-of-concepts, it is envisaged in the future the development of a dedicated full-featured kernel-level tracer, or the modification of one of the existing tracing frameworks, for the purpose of overcoming the just mentioned limitations.

B. Detection Algorithms

The algorithm for period detection presented in this paper is still preliminary. While being effective for the experimental results gathered in this paper, the algorithm needs more extensive validation over a larger class of multimedia applications, comprising single-threaded and multi-threaded applications, and especially commercial legacy software largely used in multimedia streaming, such as QuickTime$^{(TM)}$ or others.

Concerning the appropriateness of the period detection algorithm, the type of considered events needs a much deeper investigation. For example, it should be checked if by reducing or extending the set of system calls intercepted by `qostrace` an improvement of the algorithm precision, at equal observation time-window, may be obtained.

Also, an extensive evaluation of the impact of the f_{min}, f_{max}, δf, K, and Mparameters on the algorithm precision and overhead needs to be performed.

The feedback-based controller used in this paper is also subject to a variety of improvements, the first one being the type of "probe" the feedback-based control loop is based upon. In fact, the boolean information about the CBS deadline having been post-poned or not constitutes a very rough information about how tight the budget in use fits the actual application requirement. Improvements in this direction may be done by exploiting information about the actual execution-time of the reserved threads, as logged by the kernel and available, for example, by means of the `clock_gettime()` system-call via the `CLOCK_THREAD_CPUTIME_ID` clock. Alternatively, specific functions made available by the real-time scheduling framework might be exploited. For example, the AQuoSA [27] real-time scheduler implementing Hard CBS reservations, provides the (`qres_get_time()`) function for the purpose of allowing applications to read how much budget was actually consumed by the set of threads attached to the reservation (as a whole, without any need to query for the individual threads).

With the possibility to directly read the amount of budget actually consumed within the reservation, it would be much easier to decide what budget to assign for the future application jobs. For example, a simple maximum over a moving window of last observed values of consumed budget, or a percentile estimation of the budget consumption distribution, like done in [28], would constitute valuable approached to experiment with.

C. Multi-thread applications

The experimental results presented in this paper are limited to simple applications with a unique evident periodicity, while it is planned to experiment with more complex applications, possibly composed of multiple concurrent threads with possibly different periodicity. For example, a multimedia application with multiple threads (such as the VideoLAN Client - VLC[9]), one dedicated to loading the video from the disk (or receiving it from the network), one to video processing and one to audio processing, may possess a different periodicity for the three threads.

VII. 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 technologies

[9]More information is available at http://www.videolan.org/vlc/.

whose concurrent application promises to disclose important opportunities in scheduling this type of applications. The first technology is a frequency domain analyser that uses data collected in the kernel to infer important parameters of the application (such as the execution rate). The second technology is a feedback scheduler that changes the reserved budget to track the computation requirement of the application.

Experimental results gathered on a prototype of the presented mechanism show how the two technologies combine nicely, overcoming the limitations of previous work by the same authors that simply operated at the scheduler level, reacting to the changes of the some scheduling parameters. The work presented here is intended as a first step toward a more complete and general approach. We have hinted to some of the directions that we will take in carrying out our research activity. The most important contribution of this paper is that our preliminary collection of data done with a prototype implementation is very promising indeed in terms of the potential of the approach.

REFERENCES

[1] 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.

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

[3] L. Abeni, T. Cucinotta, G. Lipari, L. Marzario, and L. Palopoli, "Qos management through adaptive reservations," *Real-Time Systems Journal*, vol. 29, no. 2-3, March 2005.

[4] IEEE, *Information Technology -Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) Amendment: Additional Realtime Extensions.*, 2004.

[5] L. Abeni and L. Palopoli, "Adaptive real-time scheduling for legacy applications," in *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2008)*, Hamburg, Germany, September 2008, pp. 583–590.

[6] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son, "Design and evaluation of a feedback control edf scheduling algorithm," in *Proceedings of the 20th IEEE Real-Time Systems Symposium*, Phoenix, AZ, December 1999.

[7] 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.

[8] 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*, vol. 23, no. 1/2, September 2002.

[9] C. C. Wüst, L. Steffens, W. F. Verhaegh, R. J. Bril, and C. Hentschel, "Qos control strategies for high-quality video processing," *Real-Time Syst.*, vol. 30, no. 1-2, pp. 7–29, 2005.

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

[11] S. Banachowski and S. Brandt, "The best desktop soft real-time scheduler," in *Work-in-Progress session of the Real-Time Systems Symposium (RTSS 2001)*, London, December 2001.

[12] S. Brandt, S. Banachowski, C. Lin, and T. Bisson, "Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes," in *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*, 2003, pp. 396–407.

[13] C. Lin and S. Brandt, "Efficient soft real-time processing in an integrated system," in *Work in Progress Proceedings of the 25th IEEE Real-Time Systems Symposium (RTSS WIP 2004)*.

[14] D. Gerhard, D. Gerhard, and D. Gerhard, "Pitch extraction and fundamental frequency: History and current techniques," Tech. Rep., 2003.

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

[16] 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.

[17] G. Lipari and E. Bini, "A methodology for designing hierarchical scheduling systems," *Journal of Embedded Computing*, vol. 1, no. 2, 2004.

[18] 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*, 1989, pp. 166–171.

[19] A. K. Mok and X. A. Feng, "Towards compositionality in real-time resource partitioning based on regularity bounds," in *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 2001, p. 129.

[20] 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*. Washington, DC, USA: IEEE Computer Society, 2002, p. 173.

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

[22] L. Abeni and G. Lipari, "Implementing resource reservations in linux," in *Proceedings of Fourth Real-Time Linux Workshop*, Boston, MA, December 2002.

[23] M. Desnoyers and M. R. Dagenais, "The lttng tracer: A low impact performance and behavior monitor for GNU/Linux," in *Proceedings of the Ottawa Linux Symposium (OLS 2006)*, July 2006, pp. 209–224.

[24] M. Desnoyers, "Lttng, filling the gap between kernel instrumentation and a widely usable kernel tracer," Linux Foundation Collaboration Summit, April 2009.

[25] R. McGrath, "Utrace," Linux Foundation Collaboration Summit, April 2009.

[26] J. Keniston, "Uprobes: User space probes," Linux Foundation Collaboration Summit, April 2009.

[27] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "AQuoSA — adaptive quality of service architecture," *Software – Practice and Experience*, vol. 39, no. 1, pp. 1–31, 2009.

[28] L. Palopoli, L. Abeni, T. Cucinotta, G. Lipari, and S. K. Baruah, "Weighted feedback reclaiming for multimedia applications," in *Proceedings of the* 6^{th} *IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTImedia 2008)*, Atlanta, Georgia, United States, October 2008, pp. 121–126.