Low-Latency Audio on Linux by Means of Real-Time Scheduling [*]

Tommaso CUCINOTTA and **Dario FAGGIOLI** and **Giacomo BAGNOLI**
Scuola Superiore Sant'Anna
Via G. Moruzzi 1, 56124, Pisa (Italy)
{t.cucinotta, d.faggioli}@sssup.it, g.bagnoli@asidev.com

Abstract

In this paper, we propose to use resource reservations scheduling and feedback-based allocation techniques for the provisioning of proper timeliness guarantees to audio processing applications. This allows real-time audio tasks to meet the tight timing constraints characterizing them, even if other interactive activities are present in the system. The JACK sound infrastructure has been modified, leveraging the real-time scheduler present in the Adaptive Quality of Service Architecture (AQuoSA). The effectiveness of the proposed approach, which does not require any modifiction to existing JACK clients, is validated through extensive experiments under different load conditions.

Keywords

JACK, real-time, scheduling, time-sensitive, resource reservation

1 Introduction and Related Work

There is an increasing interest in considering General Purpose Operating Systems (GPOSes) in the context of real-time and multimedia applications. In the Personal Computing domain, multimedia sharing, playback and processing requires more and more mechanisms allowing for low and predictable latencies even in presence of background workloads nearly saturating the available resources, e.g., network links and CPU power. In the professional multimedia domain, spotting on stages, it is becoming quite common to see a digital keyboard attached to a common laptop running GNU/Linux. DJs and VJs are moving to computer based setups to the point that mixing consoles have turned from big decks into simple personal computers, only containing audio collections and running the proper mixing software.

In fact, developing complex multimedia applications on GNU/Linux allows for the exploitation of a multitude of OS services (e.g., networking), libraries (e.g., sophisticated multimedia compression libraries) and media/storage support (e.g., memory cards), as well as comfortable and easy-to-use programming and debugging tools. However, contrarily to a Real-Time Operating System (RTOS), a GPOS is not generally designed to provide scheduling guarantees to the running applications. This is why either large amount of buffering is very likely to occur, with an unavoidable impact on response time and latencies, or the POSIX fixed-priority (e.g., `SCHED_FIFO`) real-time scheduling is utilized. This turns out to be difficult when there is *more than one* time-sensitive application in the system. Though, on a nowadays GNU/Linux system, we may easily find a variety of applications with tight timing constraints that might benefit from precise scheduling guarantees, in order to provide near-professional quality of the user experience, e.g., audio acquisition and playback, multimedia (video, gaming, etc.) display, video acquisition (`v4l2`), just to cite a few of them. In such a challenging scenario in which we can easily find a few tens of threads of execution with potentially tight real-time requirements, an accurate set-up of real-time priorities may easily become cumbersome, especially for the user of the system, who is usually left alone with such critical decisions as setting the real-time priority of a multimedia task.

More advanced scheduling services than just priority based ones have been made available for Linux during the latest years, among the others by [Palopoli et al., 2009; Faggioli et al., 2009; Checconi et al., 2009; Anderson and Students, 2006; Kato et al., 2010]. Such scheduling policies are based on a clear specification that needs to be made by the application about what is the

[*] 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 n.248465 "S(o)OS – Service-oriented Operating Systems."

computing power it needs and with what time granularity (determining the latency), and this scheme is referred to as *resource reservations.* This is usually done in terms of a reservation *budget* of time units to be guaranteed every *period.* The reservation period may easily be set equal to the minimum activation period of the application. Identifying the reservation budget may be a more involved task, due to the need for a proper benchmarking phase of the application, and it is even worse in case of applications with significant fluctuations of the workload (such as it often happens in multimedia ones). Rather, it is more convenient to engage *adaptive reservation* scheduling policies, where the scheduling parameters are dynamically changed at runtime by an application-level control-loop. This acts by monitoring some application-level metrics and increasing or decreasing the amount of allocated computing resources depending on the instantaneous application workload. Some approaches of this kind are constituted by [Segovia et al., 2010; Abeni et al., 2005; Nahrstedt et al., 1998], just to mention a few.

1.1 Contribution of This Paper

This work focuses on how to provide enhanced timeliness guarantees to low-latency real-time audio applications on GNU/Linux. We use adaptive reservations within the JACK audio framework, i.e., we show how we modified JACK in order to take advantage of AQuoSA [Palopoli et al., 2009], a software architecture we developed for enriching the Linux kernel with resource reservation scheduling and adaptive reservations. Notably, in the proposed architecture, JACK needs to be patched, but audio applications using it do not require to be modified nor recompiled. We believe the discussion reported in this paper constitutes a valuable first-hand experience on how it is possible to integrate real-time scheduling policies into multimedia applications on a GPOS.

2 JACK: Jack Audio Connection Kit

JACK [1] is a well-known low-latency audio server for POSIX conforming OSes (including Linux) aiming at providing an IPC infrastructure for audio processing where sound streams may traverse multiple independent processes running on the platform. Typical applications — i.e., *clients* — are audio effects, synthesisers, samplers, tuners, and many others. These clients run as independent system processes, but they all must have an audio processing thread handling the specific computation they make on the audio stream in real-time, and using the JACK API for data exchanging.

On its hand, JACK is in direct contact with the audio infrastructure of the OS (i.e., ALSA on Linux) by means of a component referred to as (from now on) the *JACK driver* or just the *driver.* By default, *double-buffering* is used, so the JACK infrastructure is required to process audio data and filling a buffer, while the underlying hardware is playing the other one. Each time a new buffer is not yet available in time, JACK logs the occurrence of an *xrun* event.

3 AQuoSA Resource Reservation Framework

The Adaptive Quality of Service Architecture (AQuoSA[2]) is an open-source framework enabling soft real-time capabilities and QoS support in the Linux kernel. It includes: a deadline-based real-time scheduler; temporal encapsulation provided via the CBS [Abeni and Buttazzo, 1998] algorithm; various adaptive reservation strategies for building feedback-based scheduling control loops [Abeni et al., 2005]; reclamation of unused bandwidth through the SHRUB [Palopoli et al., 2008] algorithm; a simple hierarchical scheduling capability which allows for Round Robin scheduling of multiple tasks inside the same reservation; a well-designed admission-control logics [Palopoli et al., 2009] allowing controlled access to real-time scheduling capabilities of the system for unprivileged applications. For more details about AQuoSA, the reader is referred to [Palopoli et al., 2009].

4 Integrating JACK with AQuoSA

Adaptive reservations have been applied to JACK as follows. In JACK, an entire graph of end-to-end computations is activated with a periodicity equal to $\frac{buffersize}{samplerate}$ and it must complete within the same period. Therefore, a reservation is created at the start-up of JACK, and all of the JACK clients, comprising the real-time threads of the JACK server itself (the audio “drivers”), have been attached to such reservation, exploiting the hierarchical capability of AQuoSA. The reservation period has been set

[1]Note that the version 2 of JACK is used for this study

[2]More information is available at: `http://aquosa.sourceforge.net`.

equal to the period of the JACK work-flow activation. The reservation budget needs therefore to be sufficiently large so as to allow for completion of all of the JACK clients within the period, i.e., if the JACK graph comprises n clients, the execution time needed by all of the JACK clients are $c_1, \dots c_n$, and the JACK period is T, then the reservation will have the following budget Q and period P:

$$\begin{cases} Q &= \sum_{i=1}^{n} c_i \\ P &= T \end{cases} \tag{1}$$

Beside this, an AQuoSA QoS control-loop was used for controlling the reservation budget, based on the monitoring of the budget actually consumed at each JACK cycle. The percentile estimator used for setting the budget is based on a moving window of a configurable number of consumed budget figures observed in past JACK cycles, and it is tuned to estimate a configurable percentile of the consumed budget distribution (such value needs to be sufficiently close to 100%). However, the actual allocated budget is increased with respect to the results of this estimation by a (configurable) *over-provisioning* factor, since there are events that can disturb the predictor, making it potentially consider inconsistent samples, and thus nullify all the effort of adding QoS support to JACK, if not properly addressed. Examples are an xrun event and the activation of a new client, since in such case no guess can be made about the amount of budget it will need. In both cases, the budget is bumped up by a (configurable) percentage, allowing the predictor to reconstruct its queue using meaningful samples.

4.1 Implementation Details

All the AQuoSA related code is contained in the `JackAquosaController` class. The operations of creating and deleting the AQuoSA reservation are handled by the class constructor and destructor, while operations necessary for feedback scheduling — i.e., collect the measurements about used budget, managing the samples in the queue of the predictor, set new budget values, etc. — are done by the `CycleBegin` method, called once per cycle in the real-time thread of the server. Also, the `JackPosixThread` class needed some modifications, in order to attach real-time threads to the AQuoSA reservation when a new client registers with JACK, and perform the corresponding detach operation on a client termination.

The per-cycle consumed CPU time values were used to feed the AQuoSA predictor and apply the control algorithm to adjust the reservation budget.

5 Experimental Results

The proposed modifications to JACK have been validated through an extensive experimental evaluation conducted over the implemented modified JACK running on a Linux system. All experiments have been performed on a common consumer PC (Intel(R) E8400@3.00 GHz) with CPU dynamic voltage-scaling disabled, and with a Terratec EWX24/96 PCI sound card. The modified JACK framework and all the tools needed in order to reproduce the experiments presented in this section are available on-line [3].

In all the conducted experiments, results have been gathered while scheduling JACK using various scheduling policies:

- `CFS`: the default Linux scheduling policy for best effort tasks;
- `FIFO`: the Linux fixed priority real-time scheduler;
- `AQuoSA`: the AQuoSA resource reservation scheduler, without reclaiming capabilities;
- `SHRUB`: the AQuoSA resource reservation scheduler with reclaiming capabilities.

The metrics that have been measured throughout the experiments are the following:

- **audio driver timing**: the time interval between two consecutive activations of the JACK driver. Ideally it should look like an horizontal line corresponding to the value: $\frac{buffersize}{samplerate}$;
- **driver end date**: the time interval between the start of a cycle and the instant when the driver finishes writing the processed data into the sound card buffer. If this is longer than the server period, then an xrun just happened.

When the AQuoSA framework is used to provide QoS guarantees, we also monitored the following values:

- **Set budget** (**Set Q**): the budget dynamically set for the resource reservation dedicated to the JACK real-time threads;

[3] `http://retis.sssup.it/~tommaso/papers/lac11/`

- **Predicted budget** (**Predicted Q**): the value predicted at each cycle for the budget by the feedback mechanism;

Moreover, the **CPU Time** used, at each cycle, by JACK and all its clients has been measured as well. If AQuoSA is used and such value is greater than the Set Q, then an xrun occurs (unless the SHRUB reclaiming strategy is enabled).

First of all the audio driver timing in a configuration where no clients were attached to JACK has been measured, and results are shown in Table 1. JACK was using a buffer-size of 128 samples and a sample-rate of 96 kHz, resulting in a period of $1333\mu s$. Since, in this case, no other activities were running concurrently (and since the system load was being kept as low as possible), the statistics reveal a correct behaviour of all the tested scheduling strategies, with CFS exhibiting the highest variability, as it could have been expected.

Table 1: Audio driver timing of JACK with no clients using the 4 different schedulers (values are in μs).

	Min	Max	Average	Std. Dev
CFS	1268	1555	1342.769	3.028
FIFO	1243	1423	1333.268	2.421
AQuoSA	1279	1389	1333.268	2.704
SHRUB	1275	1344	1333.268	2.692

5.1 Concurrent Experiments

To investigate the benefits of using reservations to isolate the behaviour of different —concurrently running— real-time applications, a periodic task simulating the behaviour of a typical real-time application has been added to the system. The program is called `rt-app`, and it is able to execute for a configurable amount of time over some configurable period.

The scheduling policy and configuration used for JACK and for the `rt-app` instance in the experiments shown below are given in Table 2.

In all of the following experiments, we used a "fake" JACK client, `dnl`, constituted by a simple loop taking about 7% of the CPU for its computations. The audio processing pipeline of JACK is made up of 10 `dnl` clients, added one after the other. This leads to a total of 75% CPU utilisation. When AQuoSA is used (i.e., in cases (4) and (5)), JACK and all its clients share the same reservation, the budget of which is decided as described in Section 4. Concerning `rt-app`, when it is scheduled by AQuoSA

Table 2: Scheduling policy and priority (where applicable) of JACK and `rt-app` in the experiments in this section

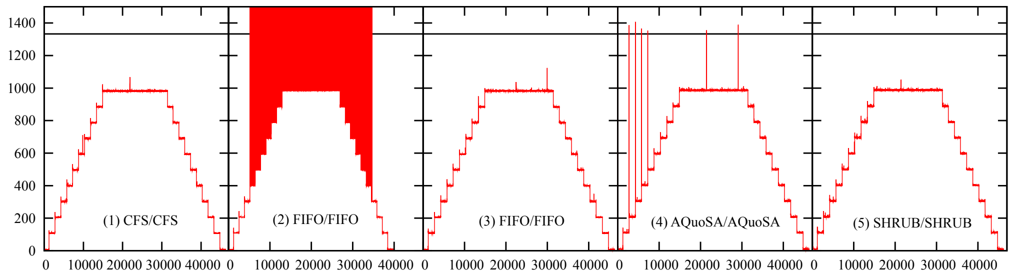
	scheduling class		priority	
	JACK	rt-app	JACK	rt-app
(1)	CFS	CFS	–	–
(2)	FIFO	FIFO	10	15
(3)	FIFO	FIFO	10	5
(4)	AQuoSA	AQuoSA	–	–
(5)	SHRUB	SHRUB	–	–

or SHRUB, the reservation period is set equal to the application period, while the budget is slightly over-provisioned with respect to its execution time (5%). Each experiment was run for 1 minute.

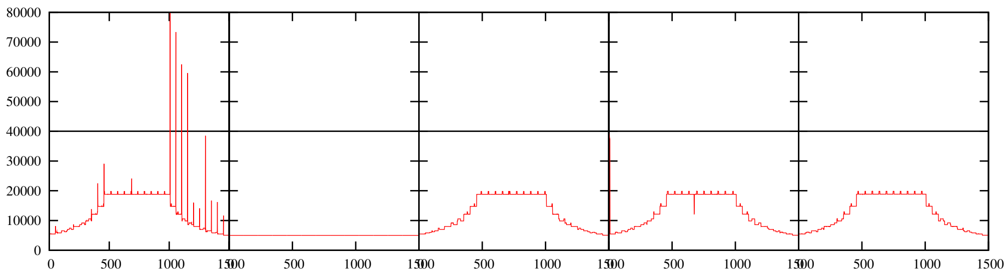
5.1.1 JACK with a period of $1333\mu s$ and video-player alike load

In this experiment, JACK is configured with a sample-rate of 96 kHz and a buffer-size of 128 samples, resulting in an activation period of $1333\mu s$, while `rt-app` has period of $40ms$ and execution time of $5ms$. This configuration for `rt-app` makes it resemble the typical workload produced by a video (e.g., MPEG format) decoder/player, displaying a video at 25 frames per second.

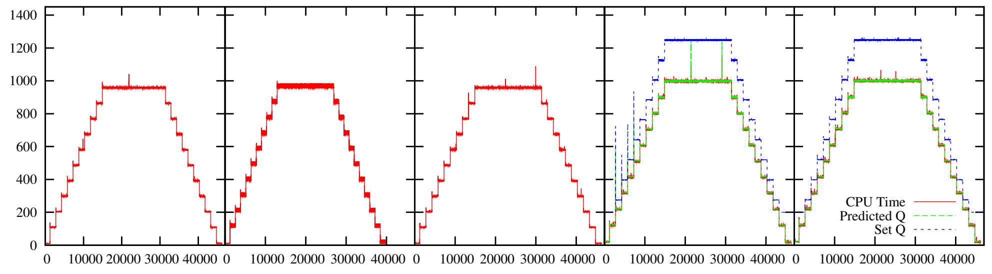
Figures 1a and 1b show the performance of JACK, in terms of driver end time, and of `rt-app`, in terms of response time, respectively. Horizontal lines at $1333\mu s$ and at $40ms$ are the deadlines. The best effort Linux scheduler manages to keep the JACK performance good, but `rt-app` undergoes increased response-times and exhibits deadline misses in correspondence of the start and termination of JACK clients. This is due to the lack of true temporal isolation between the applications (rather, the Linux CFS aims to be as fair as possible), that causes `rt-app` to miss some deadlines when JACK has peaks of computation times. The Linux fixed-priority real-time scheduler is able to correctly support both applications, but only if their relative priorities are correctly set, as shown by insets 2 and 3 (according to the well-known rate-monotonic assignment, in this case `rt-app` should have lower priority than JACK). On the contrary, when using AQuoSA (inset 4), we achieve acceptable response-times for both applications: `rt-app` keeps its finishing time well below its deadline, whilst the JACK pipeline has sporadic terminations slightly beyond the deadline, in correspondence of the registration



(a)



(b)



(c)

Figure 1: Driver end time of JACK (a) and response-times of `rt-app` (b). The Y axis reports time values in μs, while the X axis reports the application period (activation) number. The various insets report results of the experiment run under configurations (1), (2), (3), (4) and (5), from left to right, as detailed in Table 2. In (c), we report the CPU Time and (in insets 4 and 5) the set and predicted budgets for JACK during the experiment.

of the first few clients. This is due to the over-provisioning and the budget pump-up heuristics which would benefit of a slight increase in those occasions (a comparison of different heuristics is planned as future work). However, it is worth mentioning that the JACK performance in this case is basically dependent on itself only, and can be studied in isolation, independently of what else is running on the system. Finally, when enabling reclaiming of the unused bandwidth via `SHRUB` (inset 5), the slight budget shortages are compensated by the reclaiming strategy: the small budget residuals which remain unused by one of the real-time applications at each cycle are immediately reused by the other, if needed.

For the sake of completeness, Figure 1c shows the CPU Time and, for the configurations using AQuoSA, the Set Q and Predicted Q values for the experiment. The figure highlights that the over-provisioning made with a high overall JACK utilisation is probably excessive with the current heuristic, so we are working to improve it.

5.1.2 JACK with a period of $2666\mu s$ and VoIP alike load

Another experiment, very similar to the previous one but with slightly varied parameters for the two applications has been run. This time JACK has a sample-rate of $48kHz$ and a buffer-size of 128 samples, resulting in a period of $2666\mu s$, while `rt-app` has a period of $10ms$ and an execution time of $1.7ms$. This could be representative of a VoIP application, or of a 100 Hz video player.

Results are reported in Figure 5. Observations similar to the ones made for the previous experiment may be done. However, the interferences between the two applications are much more evident, because the periods are closer to each other than in the previous case. Moreover, the benefits of the reclaiming logic provided by SHRUB appears more evident here, since using just a classical hard reservation strategy (e.g., the hard CBS implemented by `AQuoSA` on the 4th insets) is not enough to guarantee correct behaviour and avoid deadline misses under the highest system load conditions (when all of the `dnl` clients are active).

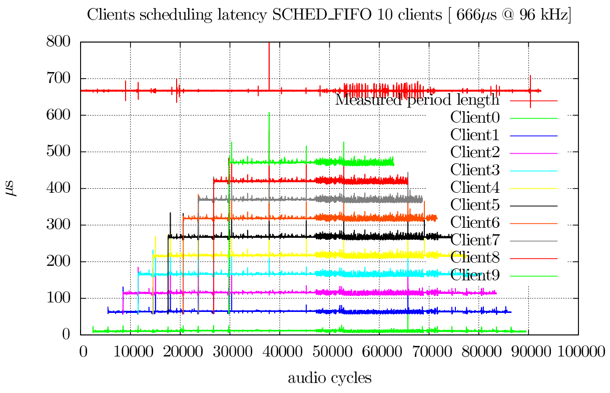


Figure 2: Server period and clients end time of JACK with minimum latency scheduled by `CFS`.

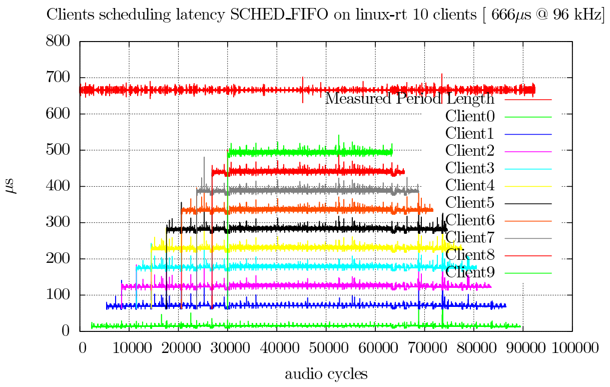


Figure 3: Server period and clients end time of JACK with minimum possible latency scheduled by `FIFO`.

5.1.3 JACK alone with minimum possible latency

Finally, we considered a scenario with JACK configured to have only 64 samples as buffer-size and a sample-rate of $96kHz$, resulting in $667\mu s$ of period. This corresponds to the minimum possible latency achievable with the mentioned audio hardware. When working at these small

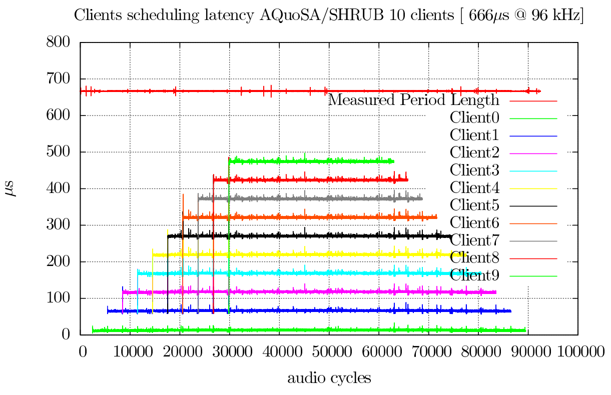


Figure 4: Server period and clients end time of JACK with minimum possible latency scheduled by `SHRUB` (reservation period was $2001\mu s$, i.e., three times the JACK period).

	`SHRUB`	`FIFO`	`CFS`
Min.	650.0	629.0	621.0
Max.	683.0	711.0	1369.0
Average	666.645	666.263	666.652
Std. Dev	0.626	1.747	2.696
Drv. End Min.	6.0	6.0	5.0
Drv. End Max.	552.0	602.0	663.0

Table 3: **period** and **driver end time** in the 3 cases (values are in μs).

values, even if there are no other applications in the system and the overall load if relatively low, xruns might occur anyway due to system overheads, resolution of the OS timers, unforeseen kernel latencies due to non-preemptive sections of kernel segments, etc.

In Figures 2, 3 and 4, we plot the **client end times**, i.e., the completion instants of each client for each cycle (relative to cycle start time). Such metric provides an overview of the times at which audio calculations are finished by each client, as well as the audio period timing used as a reference. Things are working correctly if the last client end time is lower than the server period ($667\mu s$ in this case). Clients are connected in a sequential pipeline, with `Client0` being connected to the input (whose end-times are reported in the bottommost curve), and `Client9` providing the final output to the JACK output driver (whose end-times are reported in the topmost curve). Also notice that when a client takes longer to complete, the one next to it in the pipeline starts later, and this is reflected on the period duration too. Some more details about this experiments are also reported in Table 3.

6 Conclusions and future work

In this work the JACK sound subsystem has been modified so as to leverage adaptive resource reservations as provided by the AQuoSA framework. It appears quite clear that both best effort and POSIX compliant fixed priority schedulers have issues in supporting multiple real-time applications with different timing requirements, unless the user takes the burden of setting correctly the priorities, which might be hard when the number of applications needing real-time support is large enough. On the other hand, resource reservation based approaches allow each application to be configured in isolation, without any need for a full knowledge of the entire set of deployed real-time tasks on the system, and the performance of each application will depend exclusively on its own workload, independently of what else is deployed on the system. We therefore think that it can be stated that resource reservations, together with adaptive feedback-based control of the resource allocation and effective bandwidth reclamation techniques, allows for achieving precise scheduling guarantees to individual real-time applications that are concurrently running on the system, though there seems to be some space for improving the currently implemented budget feedback-control loop.

Along the direction of future research around the topics investigated in this paper, we plan to explore on the use of two recently proposed reservation based schedulers, the IRMOS [Checconi et al., 2009] hybrid EDF/FP real-time scheduler for multi-processor systems on multi-core (or multi-processor) platforms, and the SCHED_DEADLINE [Faggioli et al., 2009] patchset, which adds a new scheduling class that uses EDF to schedule tasks.

References

Luca Abeni and Giorgio Buttazzo. 1998. Integrating multimedia applications in hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December.

Luca Abeni, Tommaso Cucinotta, Giuseppe Lipari, Luca Marzario, and Luigi Palopoli. 2005. QoS management through adaptive reservations. *Real-Time Systems Journal*, 29(2-3):131–155, March.

Dr. James H. Anderson and Students. 2006. Linux Testbed for Multiprocessor Scheduling in Real-Time Systems (*LITMUS^{RT}*). `http://www.cs.unc.edu/~anderson/litmus-rt/`.

Fabio Checconi, Tommaso Cucinotta, Dario Faggioli, and Giuseppe Lipari. 2009. Hierarchical multiprocessor CPU reservations for the linux kernel. In *Proceedings of the 5th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2009)*, Dublin, Ireland, June.

Dario Faggioli, Fabio Checconi, Michael Trimarchi, and Claudio Scordino. 2009. An edf scheduling class for the linux kernel. In *Proceedings of the 11th Real-Time Linux Workshop (RTLWS 2009)*, Dresden, Germany, October.

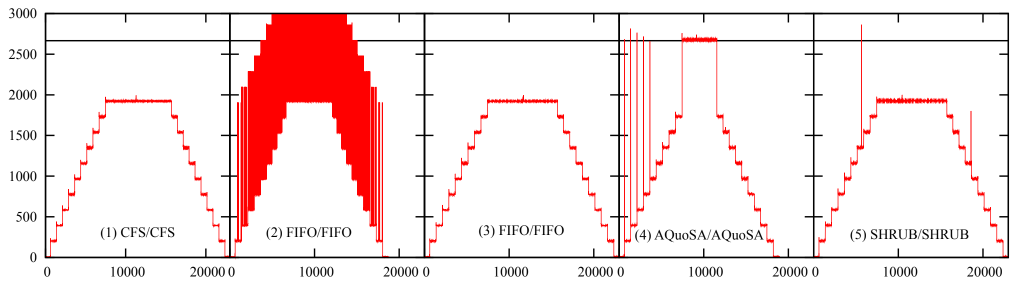
S. Kato, R. Rajkumar, and Y. Ishikawa. 2010. Airs: Supporting interactive real-time applications on multicore platforms. In *Proc. of the 22nd Euromicro Conference on Real-Time Systems (ECRTS 2010)*, Brussels, Belgium, July.

Klara Nahrstedt, Hao-hua Chu, and Srinivas Narayan. 1998. QoS-aware resource management for distributed multimedia applications. *J. High Speed Netw.*, 7(3-4):229–257.

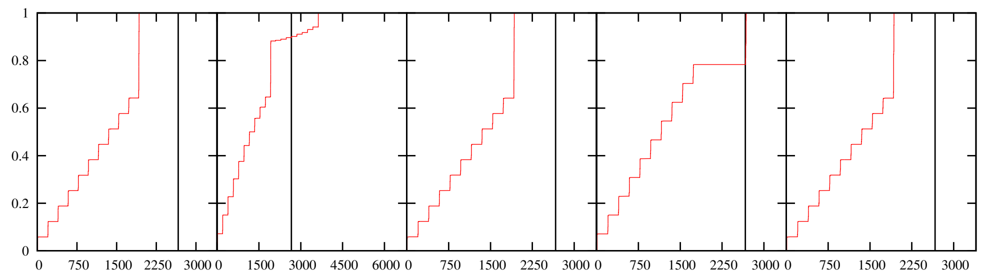
Luigi Palopoli, Luca Abeni, Tommaso Cucinotta, Giuseppe Lipari, and Sanjoy K. Baruah. 2008. Weighted feedback reclaiming for multimedia applications. In *Proceedings of the 6th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia 2008)*, pages 121–126, Atlanta, Georgia, United States, October.

Luigi Palopoli, Tommaso Cucinotta, Luca Marzario, and Giuseppe Lipari. 2009. AQuoSA — adaptive quality of service architecture. *Software – Practice and Experience*, 39(1):1–31.

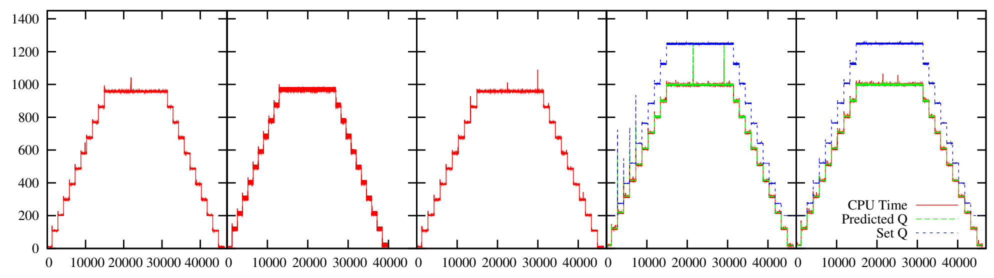
Vanessa Romero Segovia, Karl-Erik Årzén, Stefan Schorr, Raphael Guerra, Gerhard Fohler, Johan Eker, and Harald Gustafsson. 2010. Adaptive resource management framework for mobile terminals - the ACTORS approach. In *Proc. of the Workshop on Adaptive Resource Management (WARM 2010)*, Stocholm, Sweden, April.



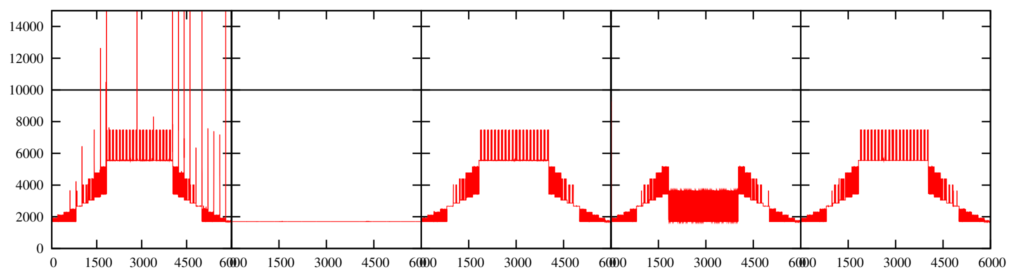
(a)



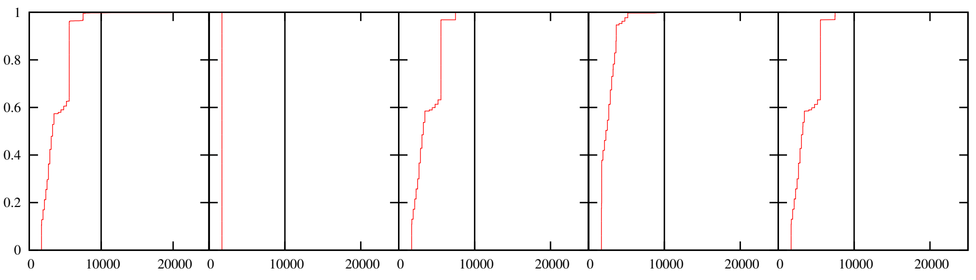
(b)



(c)



(d)



(e)

Figure 5: From top to bottom: driver end time and its CDF, JACK CPU Time and budgets, response time of `rt-app` and its CDF of the experiments with JACK and a VoIP alike load. As in Figure 1, time is in μs on the Y axes of (a)-(c)-(d), while the X axes accommodate application cycles.