

# Schedulable Device Drivers: Implementation and Experimental Results

Nicola Manica\*, Luca Abeni\*, Luigi Palopoli\*, Dario Faggioli<sup>†</sup> and Claudio Scordino<sup>‡</sup>

\*University of Trento

Trento - Italy

Email:{nicola.manica, luca.abeni, luigi.palopoli}@unitn.it

<sup>†</sup>Scuola Superiore Sant'Anna

Pisa - Italy

Email:d.faggioli@sssup.it

<sup>‡</sup>Evidence Srl

Pisa - Italy

Email:claudio@evidence.eu.com

**Abstract**—An important issue when designing real-time systems is to control the kernel latencies introduced by device drivers. This result can be achieved by transforming the interrupt handlers into schedulable entities (threads). This paper shows how to schedule such threads (using resource reservations) so that both the performance of real-time tasks and the device throughput can be controlled. In particular, some tools based on a kernel tracer (*Ftrace*) are used to collect timing information about the IRQ threads, and a novel reservation-based scheduler for Linux (*SCHED\_DEADLINE*) is used to schedule them. An implementation of the proposed technique is validated through an extensive set of experiments, using different kinds of resources and of realistic applications.

## I. INTRODUCTION

One of the prominent issues in the design of modern real-time operating systems is accounting for interference of device drivers on the real-time tasks. Indeed, the interference possibly generated by a device driver (which contributes to the so called “Kernel latencies”) introduces some unbounded blocking times that can compromise the schedulability of task sets deemed schedulable by the formal analysis techniques (because the worst case blocking times are unknown, hence it is not possible to account for them in the formal analysis).

A straightforward strategy to address this problem is to give real-time tasks higher priorities than device drivers. However, this is not possible on general purpose systems, where interrupt handlers and device drivers are not schedulable entities (and are executed with a higher priority than all the user-space tasks). For this reason, recent developments in the Linux kernel allow transforming the interrupt handlers (both Interrupt Service Routines — ISRs — and Bottom Halves) into kernel threads, the so called IRQ threads (note that in the past similar solutions have been mainly used in  $\mu$ kernel based systems [1], [2] or in proprietary real-time kernels such as LynxOS). This functionality was originally developed for the Preempt-RT real-time kernel [3], [4], and enables a control on the amount of interference from device drivers suffered by real-time tasks [5], [6], [7].

Once interrupt handlers have been transformed into schedulable entities, the problem remains open of identifying the best scheduling algorithm that can be used to serve the newly introduced threads. For example, Manica et al. [8], [9] have provided a clear evidence that using resource reservations [10] to schedule the interrupt handlers (IRQ threads, in Preempt-RT) allows the designer to find appropriate trade-offs between the response time of real-time tasks and the device throughput (this is important when the device is used by real-time tasks). However, to the best of our knowledge, most experiments and tests with advanced scheduling solutions have been performed only using prototypical schedulers or experimental Operating Systems [11], [12]. Only recently has a Linux scheduler based on resource-reservation has been proposed to the kernel community [13]. Such a scheduler exports an API that can be easily used to schedule kernel threads implementing the device drivers. Additionally, most of the previous work has focused on network devices [14], [15], [16], [5], [9] paying little or no attention to other types of devices (e.g., disks). Finally, another limitation of previous results is that they are mostly collected on artificial task sets.

This paper takes a step forward to show that the results collected with experiments based on prototypical schedulers can be repeated: 1) with a scheduler likely to become main line in the near future, 2) using different kinds of resources, 3) with realistic applications rather than with artificial task sets.

As a last contribution, a set of tools based on the *Ftrace* kernel tracing facility is used to collect the stochastic distribution of the execution time and of the inter-arrival time of device drivers. This way it is possible to apply design techniques that enable an appropriate dimensioning of the scheduling parameters.

The paper is organised as follows: Section II recalls the scheduling algorithm used in this work, briefly describes an implementation of such an algorithm, and explains how to correctly assign the scheduling parameters to IRQ threads; Section III describes the tracing tools used for analysing the scheduler behaviour and to collect information about the IRQ

threads; Section IV presents some experimental results, and Section V concludes the paper.

## II. SCHEDULING THE IRQ THREADS

This section briefly recalls some basic concepts about resource reservations, and about assigning proper reservation parameters to the interrupt threads. It also introduces the reservation-based scheduler for Linux (named `SCHED_DEADLINE`) that has been used for scheduling the interrupt threads.

### A. Reservation-Based Scheduling

The basic idea of reservation-based scheduling is that each task is reserved an amount  $Q$  of CPU time (named *maximum budget*) every  $T$  time units ( $T$  is called *reservation period*). Such a strategy can be implemented by using various scheduling algorithms. The particular reservation algorithm used in this paper is the Constant Bandwidth Server (CBS) [17], which, contrary to different scheduling choices of the same kind, is well behaved with both regular and periodic tasks and with aperiodic and dynamically changing tasks.

The CBS algorithm assigns each task a *scheduling deadline*, and schedules processes and threads using an Earliest Deadline First (EDF) policy (i.e., the task with the earliest scheduling deadline is selected first for execution). When a task wakes up, the CBS checks if its current scheduling deadline can be used; otherwise, a new scheduling deadline is generated (as  $d = t + T$ , where  $t$  is the wakeup time). The scheduling deadline is then postponed by  $T$  ( $d = d + T$ ) every time that the task executes for  $Q$  time units (if having a work conserving algorithm is not important, the task is removed from the runqueue until time  $d - T$ ).

An interesting feature of the reservation-based schedulers is that they provide *temporal isolation* among tasks. This means that the temporal behaviour of a task is not affected by the behaviour of the other tasks in the system: if a task requires a large execution time, it cannot affect the schedulability of the other tasks, or monopolise the processor. This is a basic property needed for scheduling real-time tasks on general-purpose operating systems.

### B. The Linux `SCHED_DEADLINE` Policy

In the recent versions, the official Linux kernel has introduced a new scheduling framework that replaces the old  $O(1)$  scheduler. This framework contains an extensible set of *scheduling classes*. Each scheduling class implements a specific algorithm and schedules tasks with a specific policy.

Currently, two scheduling classes are available in the Linux kernel:

- `sched_fair`, which implements the “*Completely Fair Scheduler*” (CFS) algorithm, and schedules tasks having `SCHED_OTHER` or `SCHED_BATCH` policies. Tasks are run at precise weighted speeds, so that each task receives a “fair” amount of processor share.
- `sched_rt`, which implements a POSIX fixed-priority real-time scheduler, and handles tasks having `SCHED_FIFO` or `SCHED_RR` policies.

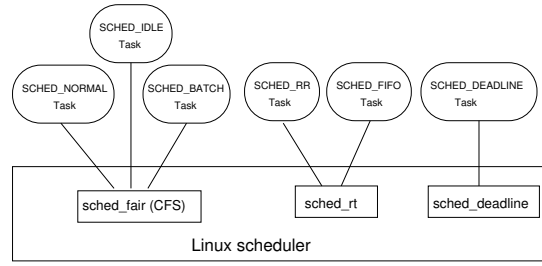


Figure 1. Linux scheduler with `SCHED_DEADLINE`.

As explained in previous papers [13], using these scheduling policies with tasks characterised by temporal constraints might be problematic, mainly because the standard API used in general purpose kernels like Linux does not allow to associate temporal constraints (e.g., deadlines) to the tasks. In fact, although it allows to assign a share of processor time to a task, there is no way to specify that the task must finish the execution of a job before a given time. Using CFS, moreover, the time elapsed between two consecutive executions of a task is not deterministic and cannot be bound, since it depends on the number of tasks running in the system at that time.

For these reasons, very recently, a new scheduling class based on resource reservations has been implemented and proposed to the kernel community<sup>1</sup>. The project, formerly known as `SCHED_EDF`, changed name to `SCHED_DEADLINE` after the request of the kernel community.

This class adds the possibility of scheduling tasks using the CBS algorithm, without changing the behaviour of tasks scheduled using the existing policies. Figure 1 depicts schematically the Linux scheduler extended with the `SCHED_DEADLINE` scheduling class (note that scheduling classes have increasing priorities from left to right).

The implementation does not make any restrictive assumption on the characteristics of the tasks. Thus, it can handle periodic, sporadic and aperiodic tasks. It is aligned with the current mainstream kernel, and it relies on standard Linux mechanisms to natively support multicore platforms and to provide hierarchical scheduling through a standard API.

a) *Main Characteristics of the Implementation:* In the implementation, red-black trees are used for ready queues to enable efficient handling of events such as earliest deadline task scheduling, new task activation, task blocking/unblocking, etc. One run-queue per each CPU is used to avoid contention and achieve high scalability even on large systems. Moreover, it is enriched with the following features:

- support for bandwidth reclaiming, to make the scheduler work conserving without affecting guarantees;
- capability of synchronising tasks with the scheduler;
- support for resource sharing similar to priority inheritance (already present in the kernel for fixed priority real-time tasks);

<sup>1</sup>`SCHED_DEADLINE`. The code is open and available at [http://gitorious.org/sched\\_deadline](http://gitorious.org/sched_deadline).

- support for standard Linux mechanisms for debugging and tracing the scheduler behaviour and for specifying per-user policies and limitations;
- capability of sending signals to the tasks on budget overruns and scheduling deadline misses;
- support for bandwidth management throughout admission control, both system-wide and for separate groups of tasks.

b) *User Level API*: An user-level application can exploit the services provided by the `SCHED_DEADLINE` scheduling class by means of some new system calls and a new data structure that accommodates additional scheduling parameters. The new data structure is called `sched_param_ex` and comprises the following fields:

- temporal parameter of the task — i.e., `sched_runtime` and `sched_deadline` which will be  $Q$  and  $T$  of its reservation, respectively;
- `sched_flag` for controlling some aspects of the scheduler behaviour. More precisely, (i) whether or not a task wants to be notified about budget overruns and/or scheduling deadline misses and (ii) whether or not a task wants to exploit some kind of bandwidth reclaiming;
- some other fields left there for backward compatibility or future extensions (`sched_priority` and `sched_period`).

The most important system calls added are:

- `sched_setscheduler_ex` (and a couple of others), that manipulates `sched_param_ex`;
- `sched_wait_interval` to synchronise the task with the scheduler. This means a task can ask to be put to sleep until either its next deadline or whenever it will be possible to receive its full budget again.

The security model adopted is very similar to the one already in use in the kernel for fixed priority real-time tasks — i.e., it is based on user permissions and capabilities and it can be affected by standard UNIX security mechanism, like `rlimits`. Controls exist for managing the fraction of CPU time usable by the whole EDF scheduler as well as to a group of EDF tasks, but they are not described here for space reasons.

### C. Assigning the Reservation Parameters

The reservation parameters  $(Q, T)$  can be dimensioned by performing a deterministic or a stochastic analysis of the interrupt behaviour [9]. The deterministic case is simpler to analyse, and allows to dimension the reservation so that no interrupt is lost (at the cost of some overestimation of the reserved CPU time). First of all, if  $P$  is the minimum inter-arrival time between two consecutive interrupts and  $C$  is the maximum amount of time needed to serve an interrupt, then  $Q$  and  $T$  must be assigned according to Equation 1

$$\frac{Q}{T} \geq \frac{C}{P} \quad (1)$$

In other words, the fraction of CPU time reserved to the IRQ thread should be  $\geq$  than the fraction of CPU time needed by the IRQ thread for executing.

However, some hardware devices have an upper bound  $N_c$  on the number of pending interrupts (interrupts that have not been processed yet), and if an interrupt fires when  $N_c$  interrupts requests are already pending, then the interrupt is lost even if Equation 1 is respected. As discussed in our previous work [9], this problem can be addressed by producing the following condition that has to be respected to avoid losing interrupts:

$$\frac{T - Q}{P} < N_c \quad (2)$$

The same paper also presented a stochastic analysis instrumental to the correct dimensioning of the CBS parameters: in this case, instead of considering the worst-case times  $P$  and  $C$ , the interrupt inter-arrival times and the execution times of the interrupt handlers are modelled as stochastic variables. As a result, the probability to drop an interrupt can be computed.

Both approaches require a precise characterisation of the workload generated by IRQ threads. Hence, the need for the tracing mechanism described in the next section.

## III. INFERRING THE IRQ PARAMETERS

According to Section II-C, if the probability distributions of the inter-arrival and execution times of an IRQ thread are known, then it is possible to schedule such thread with a  $(Q, T)$  reservation so that no interrupt is lost (note that if no interrupt is lost then the device can achieve its maximum throughput). Hence, to assign the maximum budget  $Q$  and the reservation period  $T$  to an IRQ thread it is necessary to know the *IRQ parameters* (that is, the probability distributions of the inter-interrupt times and of the times needed to serve an interrupt).

Such probability distributions can be measured by using the *Ftrace* tracer provided by the Linux kernel and by properly parsing its traces. In the proposed approach, this is done through a set of tools organised in a pipeline, as shown in Figure 2 (more details about the used tracing tools are available in a technical report [18]).

### A. The Tracing Pipeline

The kernel traces produced by *Ftrace* can be used to extract various information about tasks' timings, so that their temporal behaviour can be inferred.

The first stage of the pipeline (the trace parser) transforms the textual traces exported by *Ftrace* in an internal format, which is used by the other tools in the pipeline. This step is needed because *Ftrace* exports traces in the form of text files, whose format can change from one kernel version to another, containing redundant and unneeded information (this happens because the *Ftrace* format has been designed to be easily readable by humans). Hence, the textual traces produced by *Ftrace* are parsed and transformed in a more compact, kernel-independent, binary format which is used as input by the next stages of the pipeline. Such stages are composed by a second set of tools that can:

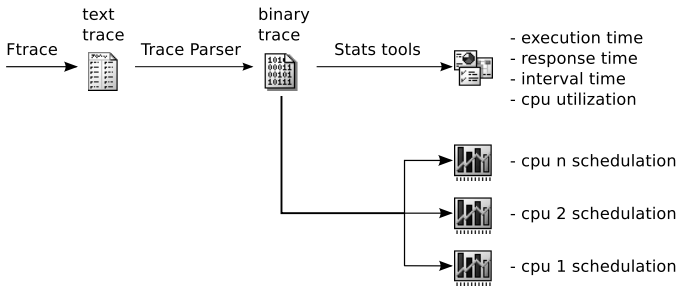


Figure 2. General architecture of the tool

- parse the internal format to gather statistics about execution times, inter-arrival times, response times, and utilisation;
- generate a chart displaying the CPU scheduling;
- infer some of the tasks temporal properties, identifying (for example) periodic tasks.

In this context, the presented tools are used to extract the probability distributions of the execution and inter-arrival times of the IRQ threads.

The various tools composing the pipeline communicate through standard Unix FIFOs (named pipes) and can be combined in different ways, to collect different kinds of information. For example, a tool which periodically displays important statistics for selected tasks (similarly to the standard “top” utility) can be inserted into the pipeline. In this work, the collected values are generally saved to files to be processed off-line later, but in other situations they can also be summarised by some statistics that are saved instead of the raw sequence of values, to save some disk space.

Since connecting the different tools in a correctly working pipeline (creating all the needed FIFOs, etc.) can sometimes be difficult, some helper scripts have been developed.

### B. Examples

The first possible usage of the proposed tools is to visually analyse the scheduler’s behaviour, to check its correctness or to understand the reason for unexpected results. An example about this usage will be presented in Section IV. If, instead, a statistics module is used in the last stage of the pipeline, it is also possible to collect some information for performance evaluation. For example, some statistics for some periodic tasks have been collected and shown in Table I. The Cumulative Distribution Functions (CDFs) of the response times for the three tasks, as measured using a different output module, are displayed in Figure 3. Note that all the results presented up to now can be obtained by just changing the final stage of the processing pipeline.

As explained, in this paper the presented tools are used to collect timing information about IRQ threads. However, before performing such measurements, it is important to test the reliability of this information. For this purpose, some experiments have been performed by considering the network IRQ threads: a stream of periodic UDP packets has been sent between two computers, measuring the inter-packet times in

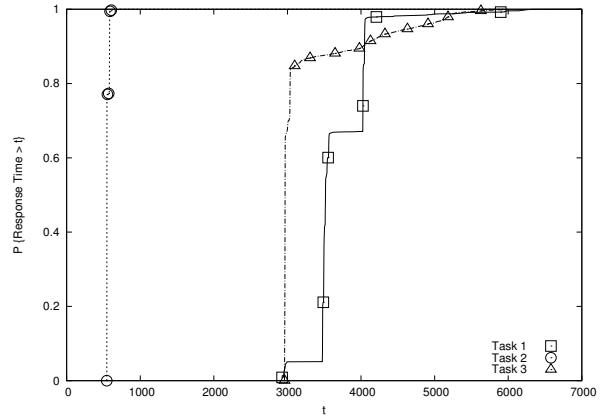


Figure 3. CDFs of the response times for 3 periodic tasks.

Table II  
INTER-PACKET TIMES AS MEASURED IN THE SENDER. TIMES ARE IN  $\mu s$ .

Test	Average	Std Dev	Max	Min
T1	1190	29	1569	1040
T5	5198	22	5278	5058
T10	10195	22	10277	10062
T50	50207	27	50298	50081
T100	100207	25	100290	100093

the sender (Table II) and in the receiver (Table III). Note that, as expected, the values in Table III almost match the values in Table II: the only noticeable difference is test T1, in which the inter-packet times on the receiver have a large maximum value and 0 as a minimum value. This is probably due to some delayed scheduling of the receiver task. After these initial measurements, the proposed tools have been used to extract the inter-arrival times of the network IRQ thread, summarised in Table IV. By comparing Table II and Table IV, it is possible to verify that the average inter-activation times of the network IRQ thread in the receiver are consistent with the average inter-packet times in the sender. The maximum times also match, while the minimum times present some differences. In particular, in tests T1, T5 and T100 the minimum inter-arrival time for the network IRQ thread in receiver is much smaller than the minimum inter-packet time in the sender. A more detailed analysis revealed that this is probably due to some non UDP packets (ICMP or ARP) which are not directly generated by the test program in the sender machine (hence, they are not periodic and they are not listed in Table II). In any case, the comparison between Table II and Table IV seems to confirm the correctness of the collected data.

Some information about the IRQ thread execution times (needed to perform some kind of performance analysis of the system) are shown in Table V, and some examples of distribution functions obtained using these tools will be presented in Section IV.

## IV. EXPERIMENTAL RESULTS

The effectiveness of the proposed approach has been tested by an extensive set of experiments. In particular, the per-

Table I  
STATISTICS COLLECTED FOR SOME PERIODIC TASKS. TIMES ARE IN  $\mu s$ .

Task	Execution Time				Inter-Arrival Time				Response Time			
	Avg	Std Dev	Max	Min	Avg	Std Dev	Max	Min	Avg	Std Dev	Max	Min
Task 1	2991	273	8953	2956	5993	303	10720	11	3182	555	5986	2960
Task 2	553	66	6025	544	2997	10	3002	2991	556	229	6027	546
Task 3	2941	51	5859	2919	7993	24	9049	6938	3683	397	7285	2927

Table III  
INTER-PACKET TIMES AS MEASURED IN THE RECEIVER. TIMES ARE IN  $\mu s$ .

Test	Average	Std Dev	Max	Min
T1	1207	1011	14336	0
T5	5212	1019	6144	4096
T10	10210	271	12288	8192
T50	50229	1023	51200	49152
T100	100204	530	100352	98304

Table IV  
INTER-ARRIVAL TIMES FOR THE NETWORK IRQ THREAD. TIMES ARE IN  $\mu s$ .

Test	Average	Std Dev	Max	Min
T1	1210	32	1424	59
T5	5222	117	5385	63
T10	10264	60	10353	10093
T50	50832	627	50353	50082
T100	100424	9342	100313	76

formance of the Linux `SCHED_DEADLINE` policy and the influence of the scheduling parameters have been evaluated when the new scheduling class is used to:

- schedule real-time tasks sets
- schedule IRQ threads
- schedule hybrid task sets composed of both real-time tasks and IRQ threads.

The next subsections will report the results obtained for each of these cases.

#### A. Using `SCHED_DEADLINE`

To see the new `SCHED_DEADLINE` policy in action, consider a periodic task (with period  $5ms$ ) and two greedy tasks (task which never block, and try to consume all the CPU time) scheduled by two CBSs ( $(1ms, 10ms)$  and  $(1ms, 4ms)$ ). Figure 4 shows a segment of the schedule produced by the tools presented in Section III.

On the other hand, Figure 5 shows how the CBS scheduler implemented by `SCHED_DEADLINE` is more effective in handling multiple time sensitive applications than the fixed priority policy `SCHED_FIFO` provided by the standard Linux kernel. A simple video player based on `GTK2` is used as a real-time task, and two player’s instances reproduce the “Big Buck Bunny”<sup>3</sup> trailer either (i) with two different `SCHED_FIFO` priority or (ii) within two CBSs, ( $12.5ms, 40ms$ ) and ( $25ms, 40ms$ ).

<sup>2</sup><http://www.gtk.org>

<sup>3</sup><http://www.bigbuckbunny.org>

Table V  
STATISTICS ABOUT THE EXECUTION TIMES OF THE IRQ THREAD. TIMES ARE IN  $\mu s$ .

Test	Average	Std Dev	Max	Min
T1	15	5	63	9
T5	19	1	68	18
T10	14	1	29	13
T50	16	2	28	15
T100	21	3	23	12

Since the video is 25 frames per second (fps), the expected time interval between two consecutive frames (named *Inter-Frame Time - IFT* - from now on) is supposed to be  $40ms$ . In this experiment, two instances of the video player are executed in parallel, using different scheduling algorithms and priorities. As it clearly emerges from the figure, when `SCHED_FIFO` is used, the player execute with higher priority correctly reproduces the stream, and the IFTs are constant around  $40ms$  (average  $39573.0\mu s$ , standard deviation  $4725.1$ ); however, the low priority instance has poor performance, to the point that the playback stops completely at frame 310 (this is the meaning of the peak in the graph) and starts again only when the other instance finished.

When the reservation-based approach enabled by `SCHED_DEADLINE` is used, instead, both the instances are able to proceed and the performance they achieve are proportional to the fraction of CPU time they can use. This is shown in the right side of the figure and by the fact that IFT average and standard deviation are, respectively,  $39082.0\mu s$ ,  $5735.3$  for  $(25, 40)$  and  $39517.0\mu s$ ,  $19455.0$  for  $(12.5, 40)$ .

#### B. Controlling the Device Throughput

The next set of experiments has been performed to check the effects of scheduling the disk IRQ thread with a  $(Q, T)$  reservation, for different values of  $Q$  and  $T$ .

First of all, the disk throughput has been measured by using the `hdparm` command and disabling the disk caches. The results of this experiment showed that the disk throughput only depends on the fraction of CPU time  $Q/T$  reserved to the disk IRQ thread, and is not affected by the specific values of  $Q$  and  $T$ . This result seems to contradict the condition expressed by Equation 2, and is probably due to the fact that the disk controller has a large buffer (i.e.,  $N_c$  is very large).

Figure 6 shows the disk throughput as a function of  $Q/T$  (confirming that the throughput is proportional to the fraction of CPU time reserved to the IRQ thread), while Figures 7 and 8 show the probability distributions of the interrupt inter-arrival and execution times. According to such

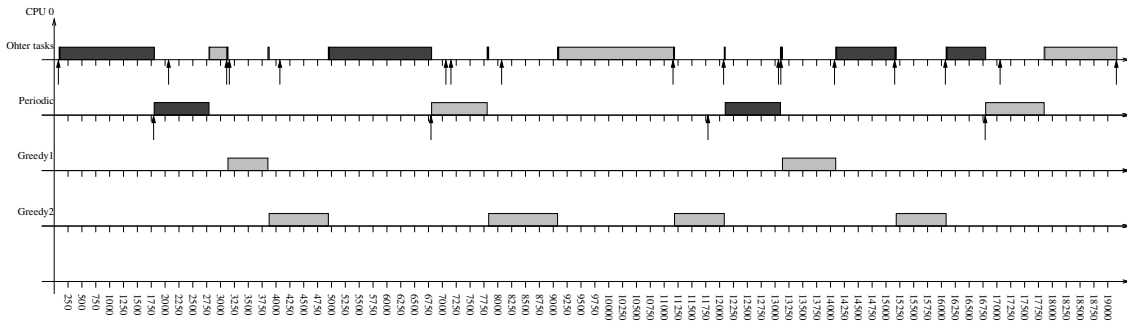


Figure 4. SCHED\_DEADLINE serving a periodic task and two CPU hungry (greedy) tasks.

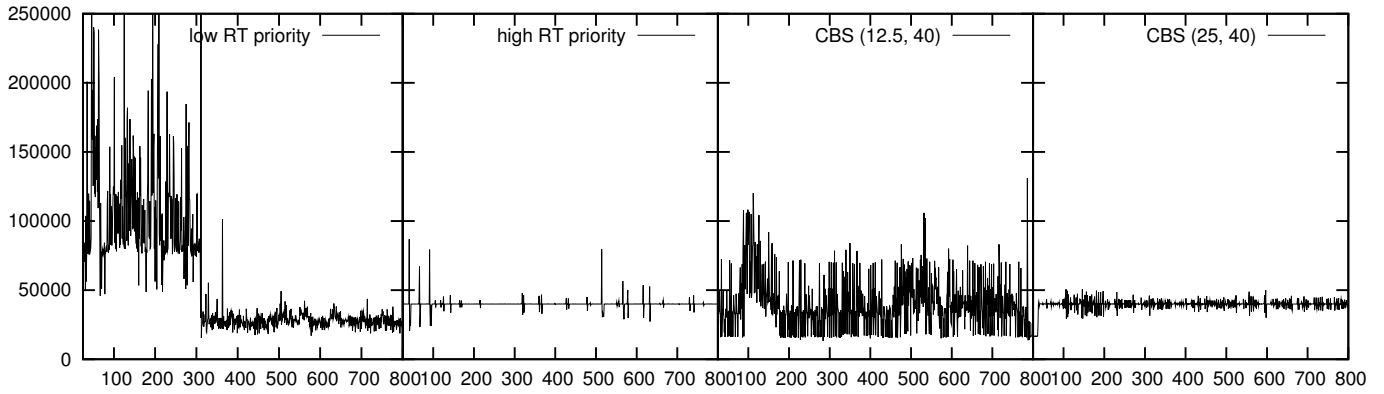


Figure 5. Inter-frame times for two instances of the video player when executing under SCHED\_FIFO (with different priorities) or SCHED\_DEADLINE. (within different reservations)

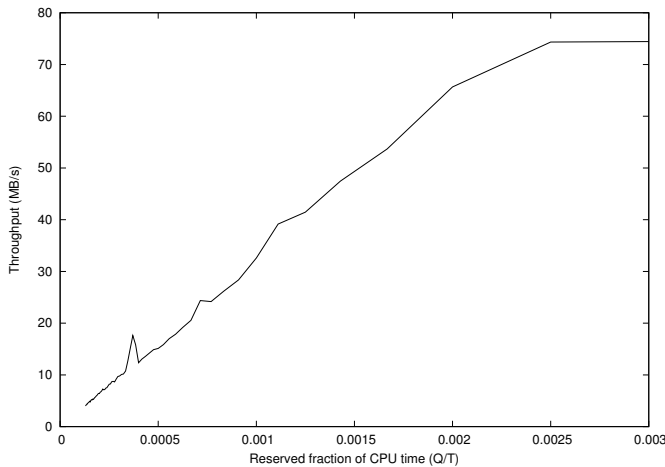


Figure 6. Disk throughput (as measured by `hdparm`) when the disk IRQ thread is scheduled by a CBS, as a function of the reserved fraction of CPU time.

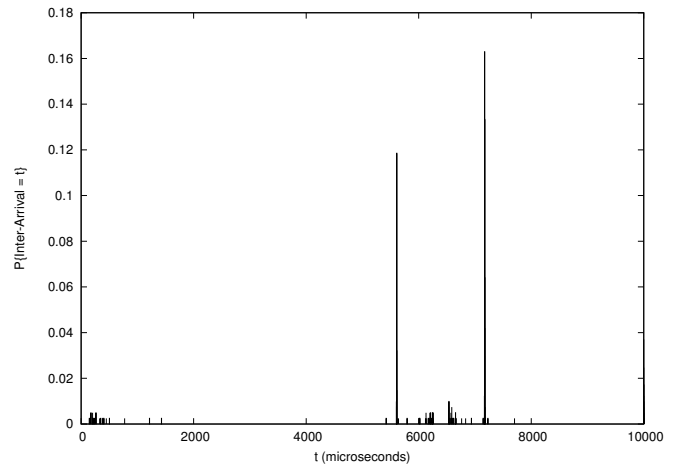


Figure 7. PMF of the inter-arrival times for the disk IRQ thread.

probability distributions, the maximum utilisation of the disk IRQ thread is about 0.41096, while the average utilisation is about 0.0021833. By comparing these data with results shown in Figure 6, it is possible to see that deterministic analysis is too pessimistic and highly overestimates the needed amount of time: in fact, `hdparm` measures the maximum

throughput<sup>4</sup> when  $Q/T = 0.003$ , which is only a little bit more than the average utilisation. By looking at Figures 7 and 8 again, it is clear that the worst case conditions (leading to the 0.41096 utilisation) are due to a long tail in the execution times probability distribution and to a small amount of small

<sup>4</sup>When running `hdparm` with the disk IRQ thread scheduled with the maximum fixed priority, the throughput resulted to be about  $75\text{MB/s}$ .

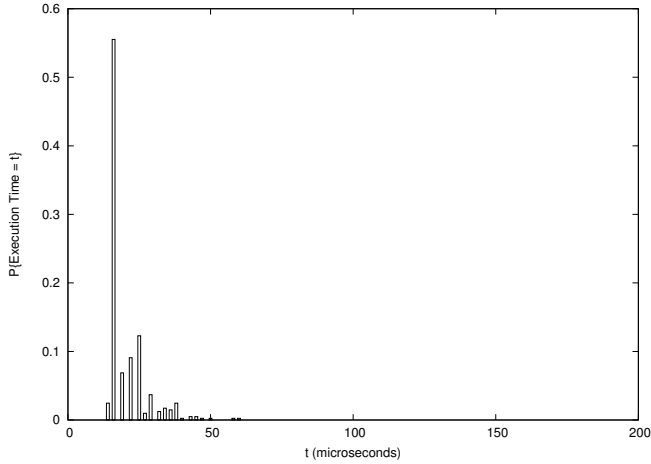


Figure 8. PMF of the execution times for the disk IRQ thread.

inter-arrival times with low probability, hence they are very unlikely. This explains why a fraction of reserved CPU time which is very close to the average utilisation is sufficient for achieving full utilisation. This consideration motivates future investigations on the application of stochastic analysis technique [9]. This research activity is currently under way.

Note that in this experiment the utilisation of disk IRQ thread is quite low, and only a small fraction of the CPU time had to be reserved to it to control the hard-disk performance. Such a low CPU utilisation caused by the disk IRQ thread is due to the usage of DMA when performing disk accesses. Such a mechanism (the DMA) allows to reduce the amount of CPU time needed by the IRQ thread, but can cause some other kind of interference (that cannot be modelled as a task in schedulability analysis) on real-time tasks due to bus contention. By disabling the DMA, all the interference is due to the IRQ thread, and can be properly accounted for in the schedulability analysis. Hence, the experiments have been repeated with DMA disabled (these experiments also allows to better understand what happens when the IRQ thread consumes more time); the results are reported in Table VI. Each line in the table is the average of the results of 20 repeated tests on a UP machine when DMA is disabled; the average utilisation for the disk IRQ thread is about 0.66, with a minimum utilisation of 0.57 and a maximum of 0.93. As expected, the throughput without DMA is much lower than the throughput achieved when using DMA, and it proportionally grows with the fraction of CPU time reserved to the interrupt thread. The maximum throughput (100% of the throughput measured when the disk IRQ thread is scheduled with a fixed priority) is achieved when  $Q/T = 0.95$ . Again, the throughput seems to only depend on the  $Q/T$  ratio, and not on the reservation period  $T$ : in other words, the average throughput achieved when using a  $(2ms, 100ms)$  reservation is the same achieved when using a  $(20ms, 1000ms)$  reservation.

After evaluating the “raw” disk performance through `hdparm`, the next experiments have been run to evaluate the performance of more complex read operations, involving

Table VI  
DISK IO-THROUGHPUT WHEN IRQ THREAD IS SCHEDULED WITH DEADLINE SCHEDULER.

Test	Average
$(2ms, 100ms)$	1.9858%
$(4ms, 100ms)$	4.23484%
$(6ms, 100ms)$	6.38374%
$(20ms, 1000ms)$	2.16843%
$(40ms, 1000ms)$	4.46488%
$(60ms, 1000ms)$	6.49811%
$(10ms, 100ms)$	10.6726%
$(20ms, 100ms)$	21.5825%
$(40ms, 100ms)$	41.8182%
$(60ms, 100ms)$	62.4476%
$(80ms, 100ms)$	82.5952%
$(90ms, 100ms)$	92.9371%
$(95ms, 100ms)$	100%
$(100ms, 1000ms)$	11.778%
$(200ms, 1000ms)$	23.261%
$(400ms, 1000ms)$	44.4056%
$(600ms, 1000ms)$	65%
$(800ms, 1000ms)$	84.5455%
$(900ms, 1000ms)$	93.007%
$(950ms, 1000ms)$	100%

Table VII  
TIME NEEDED TO PERFORM A FILE COPY, WHEN THE DISK IRQ THREAD IS SCHEDULED WITH DIFFERENT PARAMETERS.

Test	Average	Std Dev	Max	Min
No Reservations	16.89s	0.12s	17.05s	16.67s
$(30ms, 100ms)$	52.85s	0.87s	55.22s	52.36s
$(40ms, 100ms)$	39.52s	0.61s	41.25s	39.27s
$(50ms, 100ms)$	31.49s	0.12s	31.74s	31.40s
$(60ms, 100ms)$	26.23s	0.03s	26.30s	26.19s
$(70ms, 100ms)$	22.58s	0.20s	23.14s	22.47s
$(80ms, 100ms)$	19.77s	0.19s	20.31s	19.69s
$(90ms, 100ms)$	17.59s	0.04s	17.66s	17.55s

multiple system calls and file system access. The operation involved is a simple `cat` of a large file (about 44MB) redirecting output to `/dev/null`. The total time for the operation has been measured, disabling disk caches and DMA. Several runs have been repeated, and the results are reported in Table VII. The experiments were performed in a pretty old machine, and the operation lead to a very big interrupt workload, loading the CPU up to about 100% of the CPU time. The first line of the table reports the time needed for the operation when the default scheduler is used (that is, `SCHED_RT` is used for IRQ threads) in a machine with no other load. In the following lines `SCHED_DEADLINE` is used and the time falls down as the reserved fraction of CPU grows.

Note that by modifying the amount of reserved CPU time it is possible to control the amount of time needed for executing the `cat` command. In particular, the throughput (computed as the ratio between the file size and the time needed to `cat` it) is proportional to  $Q/T$ , as shown in Figure 9.

Finally, the time needed to read a large file has been analysed by measuring the system time, the user time, and the total time used by the task performing the read operation (disabling the disk caches so that the experiment is more deterministic and repeatable). The size of the file involved in this experiment was about 80MB. As expected, the total

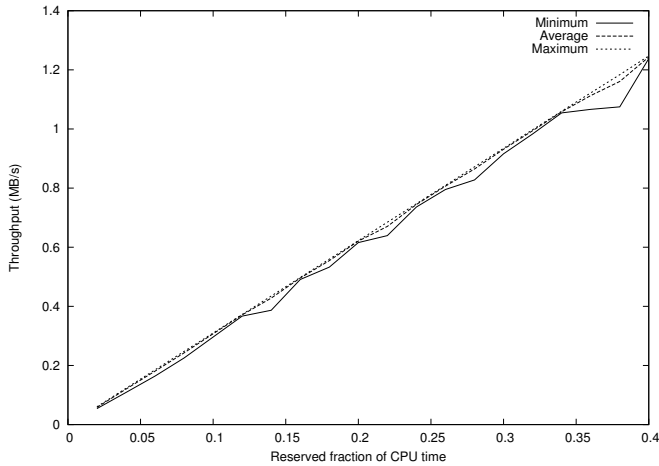


Figure 9. Throughput when reading a large file, as a function of the reserved CPU time.

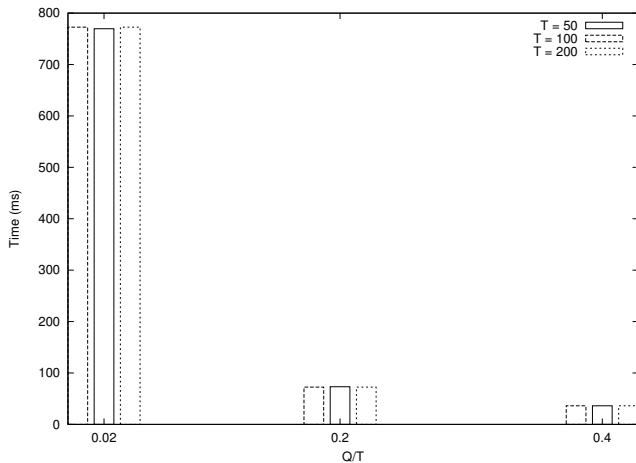


Figure 10. Total time needed to read a large file.

time needed to read the file resulted to be much larger than the sum of the system time and the user time, because the task is often blocked waiting data from the disk (so, the task performing the read operation spends most of the time in the wait state. Most of the CPU time is consumed by the disk IRQ thread, and is not visible in the statistics of the user task performing the read operation). Moreover, the amount of user time and system time used by the task resulted to be very small, and did not depend on the reservation parameters (the user time was around  $2.5ms$ , and the system time was around  $100ms$ ). On the other hand, the total time (shown in Figure 10) resulted to be proportional to  $T/Q$  (because the disk throughput is proportional to  $Q/T$ ), and (again), disk interrupts did not suffer by Equation 2.

### C. Latency/Throughput Trade-Offs

Finally, some experiments have been performed to show how the proposed approach allows one to control both the device throughput and the real-time performance of user-space tasks. The video player presented above has been used as a

Table VIII  
NETWORK THROUGHPUT ACHIEVED BY USING DIFFERENT RESERVATION PARAMETERS FOR THE VIDEO PLAYER AND FOR THE NETWORK HARD IRQ.

Test	Player CBS	net IRQ CBS	Throughput
Test1	(29ms, 40ms)	(9ms, 100ms)	59.75Mbps
Test2	(28ms, 40ms)	(12ms, 100ms)	65.43Mbps
Test3	(26ms, 40ms)	(13ms, 100ms)	70.83Mbps
Test4	(25ms, 40ms)	(14ms, 100ms)	76.14Mbps
Test5	(20ms, 40ms)	(18ms, 100ms)	88.55Mbps

real-time task, and the network device has been considered (using `netperf` to generate network load and to measure the network throughput [9]).

An instance of `netperf` has then been activated concurrently with the video player, and the experiment has been repeated scheduling the video player as `SCHED_OTHER` and as `SCHED_FIFO` with a priority higher than the network IRQ threads. The results, displayed in Figure 11, show that when the player executes alone it is able to correctly reproduce the video (the inter-frame times are constant around  $40ms$ ), while when some concurrent network load is created, the inter-frame times increase a lot (and video playback is not continuous). Finally, if the player is scheduled with a priority higher than the priority of the network IRQ threads, then it is again able to work correctly, but in this case **the network throughput measured by `netperf` drops from 88Mbps to about 58Mbps**.

A trade-off between real-time performance for the player and high network throughput can be found by using reservation-based scheduling. To this purpose, the tool presented in Section III can be used to collect the probability distributions of the execution and inter-arrival times of the network IRQ threads, and these data can be used as shown in Section II-C.

Based on such analysis, the reservation parameters shown in Table VIII have been used, obtaining the network throughput shown in the last column of the table. The evolution of the inter-frame times in the player for the most interesting cases is shown in Figure 12. As it is possible to perceive by looking at the table and at the figure, resource reservations really allow to find latency/throughput trade-offs, as previously claimed:

- if the player is reserved enough CPU time ( $29ms$  every period of  $40ms$ ), then the inter-frame times are stable and near to  $40ms$  (see the right side of Figure 12). However, in this case it is possible to reserve only a small fraction of the CPU time to the network IRQ thread, and the network throughput is low (about  $60Mbps$ );
- if enough CPU time is reserved to the network IRQ thread ( $18ms$  every  $100ms$ ), then the maximum network throughput can be achieved. But in this case it is possible to reserve only  $20ms$  of CPU time every  $40ms$  to the player, and the inter-frame times increase (see the left side of Figure 12). Note, however, that the maximum inter-frame times are still under  $80ms$  (compare this situation with the middle of Figure 11);
- some compromises can be found: for example, scheduling

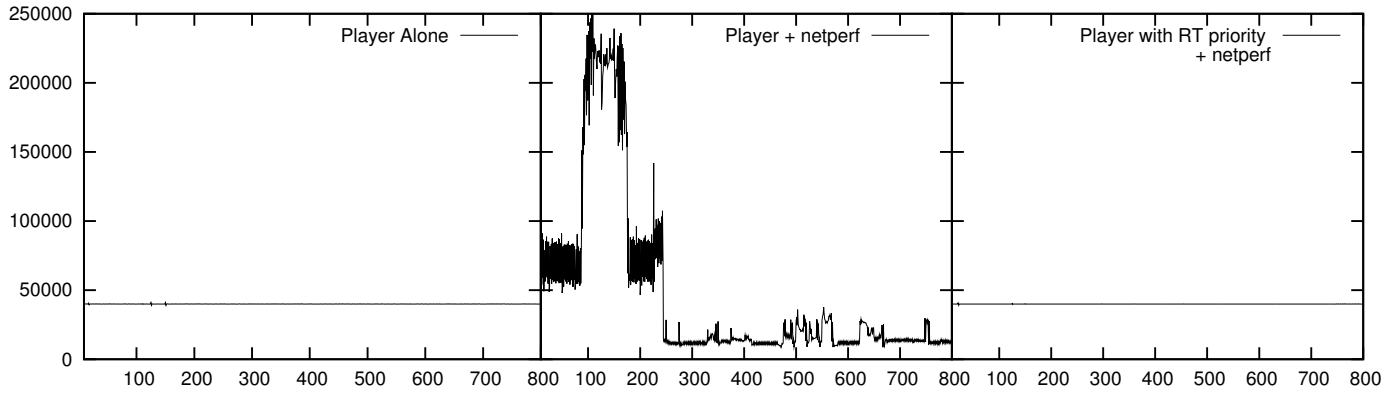


Figure 11. Inter-frame times for the video player when executed alone and concurrently with `netperf`, with different priorities.

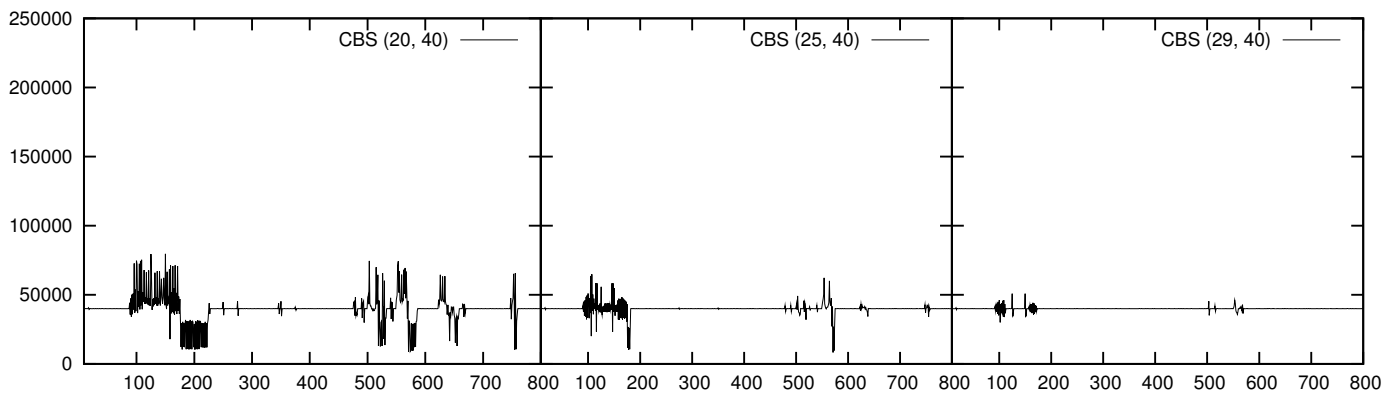


Figure 12. Inter-frame times for a video player when executed alone and concurrently with `netperf`, using different kinds of reservations.

the player with a  $(25ms, 40ms)$  reservation and the network IRQ thread with a  $(14ms, 100ms)$  reservation it is possible to have reasonable inter-frame times with a good network throughput (about 86% of the maximum).

## V. CONCLUSIONS AND FUTURE WORK

This paper reported the results of some experiences with device drivers scheduling in real-time systems. In particular, some of the presented experiments show how recent developments in the Linux kernel can be exploited to schedule the IRQ threads so that both their interference on real-time tasks and the device throughput can be controlled.

The proposed solution is based on the Linux Preempt-RT kernel (which transforms interrupt handlers into schedulable entities), a new reservation-based scheduler, and some tools based on `Ftrace` that can be used to infer the timing information needed to correctly assign the scheduling parameters.

As a future work, the effectiveness and usability of the stochastic analysis for IRQ threads will be investigated by considering different workloads and resources. This will probably require to simplify the Markov model used in the stochastic analysis, so that it can be applied more easily.

## ACKNOWLEDGEMENTS

This work has been partially supported by the European Commission under the ACTORS project (FP7-ICT-216586).

This project has also been supported by the “Provincia Autonoma di Trento”<sup>5</sup> by means of the PAT/CRS Project RoSE (<http://imedia.disi.unitn.it/RoSE>).

## REFERENCES

- [1] H. Haertig, R. Baumgartl, M. Borriss, C. Joachim Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schnberg, and J. Wolter, “DROPS - OS support for distributed multimedia applications,” in *In Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.
- [2] H. Haertig and M. Roitzsch, “Ten years of research on L4-based real-time systems,” in *Proceedings of the Eighth Real-Time Linux Workshop*, Lanzhou, China, 2006.
- [3] S. Rostedt, “Internals of the rt patch,” in *Proceedings of the Linux Symposium*, Ottawa, Canada, June 2007.
- [4] L. Henriques, “Threaded IRQs on Linux PREEMPT-RT,” in *Proceedings of Fifth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERS)*, Dublin, Ireland, June 2009.

<sup>5</sup>Translation of the original statement: “Lavoro eseguito con il contributo della Provincia autonoma di Trento”.

- [5] M. Lewandowski, M. Stanovich, T. Baker, K. Gopalan, and Wang, "Modeling device driver effects in real-time schedulability analysis: Study of a network driver," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, Bellevue, WA, 2007.
- [6] T. Baker, A. Wang, and M. J. Stanovich, "Fitting linux device drivers into an analyzable scheduling framework," in *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-time Applications*, Pisa, Italy, July 2007.
- [7] G. Modena, L. Abeni, and L. Palopoli, "Providing qos by scheduling interrupt threads," in *Work in Progress of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium, (RTAS 2008)*, St. Louis, MO, April 2008.
- [8] L. Abeni, N. Manica, and L. Palopoli, "Reservation-based scheduling for irq threads," in *Proceedings of the 11th Real-Time Linux Workshop*, Dresden, Germany, September 2009.
- [9] N. Manica, L. Abeni, and L. Palopoli, "Reservation-based interrupt scheduling," in *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium, (RTAS 2010)*, Stockholm, Sweden, April 2010.
- [10] 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.
- [11] I. 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*, vol. 14, no. 7, pp. 1280–1297, 1996.
- [12] R. Black, P. Barham, A. Donnelly, and N. Stratford, "Protocol implementation in a vertically structured operating system," in *Proceedings of the 22nd Annual Conference on Local Computer Networks*, 1997.
- [13] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino, "An EDF scheduling class for the Linux kernel," in *Proceedings of the Eleventh Real-Time Linux Workshop*, Dresden, Germany, September 2009.
- [14] K. Jeffay and G. Lamastra, "A comparative study of the realization of rate-based computing services in general purpose operating systems," in *Proceedings of the Seventh IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA)*, Cheju Island, South Korea, 2000, pp. 81–90.
- [15] S. Ghosh and R. Rajkumar, "Resource management of the OS network subsystem," in *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2002.(ISORC 2002)*, 2002, pp. 271–279.
- [16] Y. Zhang and R. West, "Process-aware interrupt scheduling and accounting," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, Rio de Janeiro, Brazil, December 2006.
- [17] 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.
- [18] P. Rallo, N. Manica, and L. Abeni, "Inferring temporal behaviours through kernel tracing," DISI - University of Trento, Tech. Rep. DISI-10-021, April 2010.