

# Reservation-Based Scheduling for IRQ Threads

**Luca Abeni**

University of Trento  
Via Sommarive 14, Povo (TN)  
luca.abeni@unitn.it

**Nicola Manica**

University of Trento  
Via Sommarive 14 Povo (TN)  
nicola.manica@gmail.com

**Luigi Palopoli**

University of Trento  
Via Sommarive 14, Povo (TN)  
palopoli@disi.unitn.it

## Abstract

Scheduling applications that require multiple hardware resources (CPU, disk, network, etc...) is known as a challenging issue in real-time systems, due to the interference from device drivers (in particular, interrupt handlers) on high-priority real-time tasks. This paper proposes a possible solution for this problem, based on scheduling interrupt handlers through a reservation-based scheduler. The solution is based on a combination of the Realtime-Preempt patch for Linux (which transforms interrupt service routines in threads) and a reservation-based scheduler, the Constant Bandwidth Server - CBS (which is used for scheduling such threads).

## 1 Introduction

Real-time scheduling theory has traditionally dealt with the problem of scheduling the CPU so that the execution of a set of concurrent tasks can meet some timing constraints. The kind of real-time constraints considered range from hard real-time constraints (requiring strict and deterministic execution guarantees) to soft real-time constraints (for which occasional violations can be tolerated and probabilistic performance guarantees are required). However, scheduling real-time applications that simultaneously use multiple hardware resources (such as CPU, disk, network, ...) is still an open issue. As one of the possible problems that arise in this context,

let us focus on the execution of device drivers. In traditional non real-time operating systems, device drivers are executed as non-preemptable code inside the kernel. Therefore, they can block the execution of real-time threads, even if the latter have in that moment a very high priority. This way the real-time guarantees are at serious risk of being disrupted.

A preliminary step to solve this problem is recognised to be making the device driver schedulable entities alongside of real-time threads. One possible way to do this (e.g., pursued in  $\mu$ kernel based systems [1, 2]) is to execute Interrupt Service Routines (ISR) inside IRQ threads. Recently a similar solution has been implemented also in the Real-time Preempt Linux kernel patch (by Ingo Molnar and oth-

ers). Some recent pieces of work have shed some light on a largely underestimated problem: **a real operating system kernel needs some CPU time to exchange data with hardware devices** [3, 5]. For instance, it is completely useless to precisely schedule a device (e.g., a disk) if the kernel is not able to find enough CPU time to manage the incoming data. And, the CPU time spent by the kernel for handling the device must not be accounted to real-time tasks that do not use such a device (causing deadline misses). This means that a really coordinated strategy for the scheduling of different resources [6] is needed.

In a previous preliminary work [7], we have shown that by selecting appropriate fixed scheduling priorities for the IRQ thread it is possible to avoid the interference of the device driver on the real-time threads, thus facilitating the enforcement of real-time constraints on their execution. In the same paper, we have shown that the price to be paid for this solution is an abrupt reduction in throughput of the IO operations. The problem is largely due to the inadequacy of a fixed priority scheduling, which for this type of applications reveals itself too coarse a tool. To get an intuition, by priorities we are only able to decide the relative importance of the tasks. This can be sufficient to say whether or not a very strictly characterised task set is able to execute concurrently meeting the time constraints [8]. If, on the contrary, we aim at a more complex goal, such as striking a good balance between real-time guarantees of a thread and throughput of an IO burst, it is not sufficient to allow or deny altogether the possibility for a device driver to steal execution time to a real-time thread. We need to control, in some sense, the amount of time that a device driver can take from a real-time thread. In other words, we need to complement the idea of making each device driver a schedulable entity with the adoption of a scheduling strategy (such as the resource reservations [10]) enabling a fine tuned control of the fraction of CPU allocated to each task.

This paper lies exactly in this line: we show how it is possible to combine the adoption of the RT patch (whereby each device driver can be executed in an IRQ thread) with a scheduling algorithm known as the Constant Bandwidth Server (CBS) [11] (which allows us to control the CPU fraction allocated to each thread). In the paper, we discuss how to carry out the integration between the RT patch and a Linux based implementation of the CBS [12]. In our experimental section we show how the combined use of

this technology offers substantial improvements over both the standard technology, and the simple use of the RT patch with fixed priorities.

## 2 The Problem

As discussed above, if some applications in the system use hardware resources other than the CPU they can “steal” execution time to high-priority real-time tasks. This happens because the OS kernel needs to consume CPU time in handling interrupts coming from the various hardware devices providing the resources, and traditional (non real-time) kernels tend to serve all the interrupt requests before scheduling user tasks (even real-time ones).

This time spent in serving interrupt requests can be a problem for real-time applications, because a low-priority, non real-time, task can affect the schedulability of a high-priority, real-time, task by performing heavy I/O (for example, a non real-time “find /” operation is known to be the source of high kernel latencies).

When a hardware interrupt fires (an Interrupt ReQuest - IRQ - arrives), it is served by a short Interrupt Service Routine (ISR, or hard IRQ handler) and by a “postponed interrupt handler”, called soft IRQ handler, which has a lower priority than the hard IRQ handler but still has a higher priority than regular threads and processes. As a result, hard and soft IRQ handlers can consume CPU time that should be devoted to more important real-time tasks. In other words, interrupt handlers can be the source of high kernel latencies<sup>1</sup>. As a consequence, a low-priority task can jeopardise the schedulability of high-priority real-time tasks by causing the generation of a large number of hardware interrupts. The standard methodology to decide the schedulability of a set of real-time tasks is by some admission tests using the total utilisation of the tasks [8], or by response time analysis [4]. There are standard ways to enhance such admission tests to account for the blocking time introduced by non-preemptive code segments and/or for the interference introduced by ISRs (modelled as high priority tasks). This way, the time “stolen” by ISRs can be accounted in the schedulability analysis.

A very popular way to reduce the impact of interrupt handlers on real-time tasks is to make the interrupt handlers subject to the scheduling policy rather than a non-preemptive section (generating

---

<sup>1</sup>The kernel latency is defined as the difference between the time when an event happens and the time when it was expected to happen.

blocking time for all tasks irrespective of their priority). For this reason, the RealTime Preemption patch (RT-preempt) [9] for the Linux kernel transforms hard and soft IRQ handlers in kernel threads (the hard IRQ thread and the soft IRQ thread), that are *schedulable entities* handled by the task scheduler in the same way as user tasks (so, IRQ threads can have lower priorities than real-time tasks, and can be preempted by them). A real-time application that does not need to interact with a specific device can schedule its tasks in foreground respect to the device's interrupt handlers, so that real-time tasks are not disturbed by the device's interrupts.

This solution can present a slightly higher overhead, and requires a more careful synchronisation, but also has the advantage of permitting to correctly account the handler code in a real-time system (that is, the CPU time required to execute the handler can be correctly accounted in order not to break the system's guarantees). The possibility to schedule interrupt handlers (provided by IRQ threads) permits to give user-space real-time tasks higher priorities than interrupts, reducing the interference from hardware devices.

Although a very important step, the possibility of scheduling interrupt threads is not a conclusive answer to the problem of providing real-time guarantees to applications with intense I/O operations. Indeed the problem remains open of how to properly assign scheduling priorities. While effective methodologies such as Rate-Monotonic or Earliest Deadline First [8] can be used for standard real-time task sets, when IRQ threads are present in the priority assignment is to be considered largely an open issue (as of today). As discussed next, an optimal priority assignment in this case could even be impossible to find.

Of course, it is easy to find priority assignments that provide good real-time performance in specific cases: for example, when real-time applications do not need to access a hardware device, the IRQ threads provided by RT-Preempt can simply be assigned a low priority reducing their interference on real-time tasks. The solution is much less obvious when some real-time activity use the data coming from the device. To expose the problem, consider a real-time application composed of periodic threads and of a thread receiving data from a hardware device. The simplest configuration revealing the problem is given by a periodic real-time thread and by a thread receiving high-bandwidth network data. In our experimental setup:

- a network card has been selected as an inter-

rupt generating device because it is easy to generate a controlled load on it, and to measure the **network** throughput;

- a periodic periodic real-time thread (scheduled using real-time - **SCHED\_FIFO** - priorities) has been used to generate some time sensitive **CPU** load;
- real-time performance have been quantified by measuring the response time of the periodic thread. Such response time is strictly related to the kernel latency [12], which represents a good measure of the real-time performance of the OS kernel, because it must be accounted for in the admission test as a *blocking time*  $B$ , so high latency values risk to make unschedulable task sets that would be schedulable if kernel effects were not considered.

The periodic real-time task has been implemented by using a program that periodically executes a busy loop (this way the computation time of each job of the task can be approximatively considered as a constant one). This program is similar to **cyclictest** program [13], but it consumes a considerable amount of time at each period, and it measures the response time of each activation. When the periodic task is the highest priority task in the system, the response times are expected to be equal to the execution times; otherwise, the interference from higher priority tasks must be considered. In this experiment, a periodic task with period  $P = 20ms$  and execution time between  $6ms$  and  $8.5ms$  has been used. The network throughput has been measured by using the **netperf** program [14].

When using the standard Fixed Priority Scheduler provided by Linux, the real-time task and the network IRQ threads have been scheduled through the **SCHED\_FIFO** policy, and various priority assignments have been tested to check their impact on the latency and on the throughput. In particular, the goal of this set of experiments was to check how manipulating the priorities of the interrupt threads allows us to control the real-time tasks' latency and the network throughput. To reduce the impact of external factors, the experimental setup is composed of two computers connected by a cross network cable. One of the two computers runs the **netperf** server, while the other one generates the network traffic by using a **netperf** client and runs the periodic real-time task.

The priority of the periodic real-time task has been fixed to 80 and the priorities of the IRQ threads serving the network card (the hard IRQ thread, and

the `softirq-net-rx` thread - these two threads will be indicated as “networking threads”) have been varied from 1 (minimum priority) to 99 (maximum priority). To better expose the effects of these two threads, `netperf` has been configured to use small UDP packets

| IRQ Priority              | < 80  | 80    | > 80    |
|---------------------------|-------|-------|---------|
| Response Time ( $\mu s$ ) | 8431  | 11284 | 1338573 |
| Throughput (Mbps)         | 66.54 | 67.42 | 81.17   |

**TABLE 1:** Response time for the Real-Time task and Network Throughput with fixed priorities

A first set of experiments indicated that assigning the same priority to the hard IRQ thread and to the soft IRQ thread gives the best results, hence the results reported in this paper are relative to priorities assigned in this way. The experiments’ results showed three different possibilities for priority assignment:

1. the networking threads have the lowest priorities in the system. This includes all the priorities from 1 to 79;
2. the networking threads have the same priority of the periodic real-time thread: 50;
3. the networking threads have a priority higher than the periodic real-time thread: this include all the priorities ranging from 81 to 99.

Table 1 summarises the results obtained in the most relevant cases. In particular, it is possible to see that when the networking threads have priority from 1 to 79 (case 1), then the worst-case response time for the periodic task is  $8.431ms$  (about the execution time of the task). On the other hand, the network throughput is low (around  $66Mbps$ ).

The response times measured in case 2 (network threads priorities equal to the real-time threads) are a little bit larger than the ones measured in case one (the worst case is  $11.284ms$ ), and the network throughput is basically unchanged.

<sup>2</sup>Note that when using 300 bytes-long UDP packets this value is near to the maximum achievable throughput

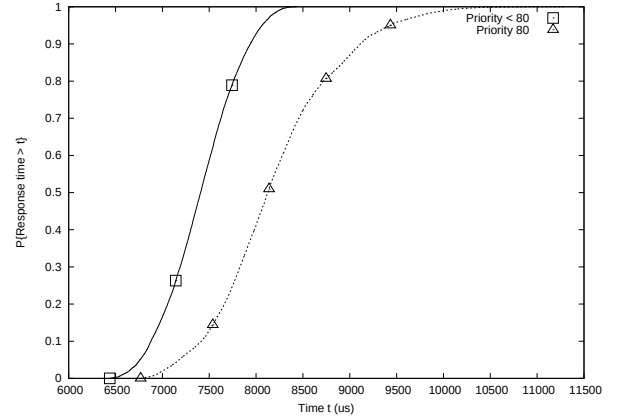


**FIGURE 1:** CDF of the finishing times for the real-time task, when IRQ threads have priority  $\leq 80$ .

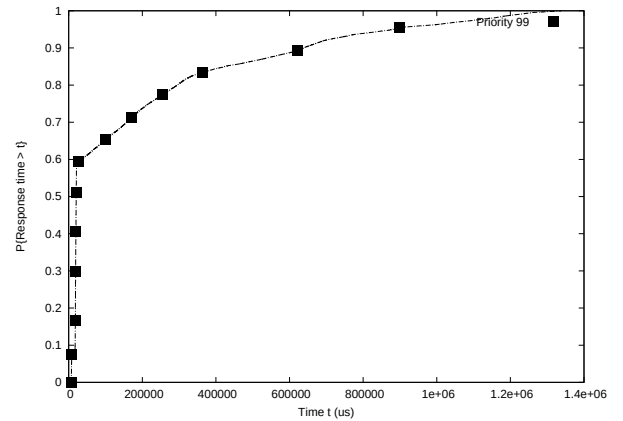


**FIGURE 2:** CDF of the finishing times for the real-time task, when IRQ threads have priority 99.

Further increasing the networking threads priority (case 3) increases the response times to more than  $1s$ , and allows to achieve a throughput of about  $81Mbps$ <sup>2</sup>. The Cumulative Distribution Function (CDF) of the real-time response times in the three cases mentioned above are reported in Figure 1 and Figure 2. From the figures it is possible to notice that when the IRQ threads priority is set to 99 the kernel latencies are completely out of control.

Unfortunately, the increase in latency is not gradual, so it is not possible to assign tasks’ priorities to obtain a response time between  $11ms$  and  $1s$ ; in the same way, it is not possible to have a fine-grained control on the network throughput by only playing with priorities.

As a concluding remark, while the introduction of the RT-patch leads to a significant improvement in real-time performance (see [7] for a comparison between the technology illustrated here and the mainstream technology in the Linux Kernel) the scheduling priorities prove unfit to find a good balance between the performance of different tasks. As discussed in the next section, this limitation can be overcome using a different scheduler).

### 3 A Possible Solution

In order to strike a good tradeoff between the performance of real-time tasks (in terms of missed deadlines) and the throughput of hardware devices, we propose reservation-based scheduling for the IRQ threads.

In particular, each IRQ thread is reserved an amount of CPU time  $Q^s$  in a reservation period  $T^s$  (this is often indicated by saying that the IRQ thread is attached to a CPU reservation  $(Q^s, T^s)$ , or is served by it);  $Q^s$  is called *maximum budget*, and  $T^s$  is called *reservation period*.

By attaching an IRQ thread to a reservation  $RSV = (Q^s, T^s)$  it is possible to enforce that such thread will not demand a fraction of the CPU time larger than  $Q^s/T^s$ . In this way, it cannot increase too much the response time of real-time tasks. If a *hard reservation* behaviour is used [10] (see the CBS description below for more details about the hard behaviour), it is also possible to guarantee that the IRQ threads will not starve non real-time applications which are running in background. In plain terms, the hard reservations behaviour implements a non-work-conserving paradigm, in which the task never receives more than reserved time<sup>3</sup>. From a more formal point of view, resource reservations allow to guarantee that the worst case behaviour of a real-time task  $\tau$  is not affected by the other tasks running in the system (thanks to the *temporal isolation* property).

Many different scheduling algorithms can be used to implement the resource reservations abstraction, and the reservation mechanism used in the paper is the Constant Bandwidth Server (CBS) [11]. A simplified description of the CBS algorithm follows: when a task  $\tau$  is served, the CPU time consumed by it is accounted by decreasing a variable  $q$  called *budget*, and tasks are scheduled based on *scheduling deadlines*  $d^s$  assigned by the CBS (the client with

the earliest scheduling deadline is scheduled). When  $\tau$  is created,  $q$  and  $d^s$  are initialised to 0. When a task  $\tau$  is activated (enters the ready queue) at time  $t$ , the CBS has to assign a scheduling deadline to it. If  $t > d^s - \frac{q}{Q^s}T^s$ , then the latest scheduling deadline  $d^s$  (and the latest budget  $q$ ) can be used; otherwise, a new scheduling deadline  $d^s = t + T^s$  is generated and the budget  $q$  is replenished to  $Q^s$ . When  $q$  arrives to 0  $\tau$  is said to be depleted, and two different behaviours are possible:

- $\tau$  is not schedulable until time  $d^s$ , when the budget will be replenished to  $Q^s$  and the deadline will be postponed to  $d^s + T^s$ . This is known as *hard reservation behaviour*.
- the budget is immediately replenished and the scheduling deadline is immediately postponed as above;  $\tau$  remains schedulable, but its priority is decreased (due to the deadline postponing). This behaviour is referred in this paper as the *soft reservation behaviour* (even if the term “firm” would be more appropriate than “soft” [10]).

For a further discussion on the CBS algorithm and on its properties, the reader is referred to the original paper [11].

Using a CBS (especially with the hard reservation behaviour) to serve IRQ threads allows to control the amount of time dedicated to device drivers, reducing their interference on real-time tasks without compromising the device throughput. To verify this property, the previous experiments have been repeated using some CBSs to schedule the IRQ threads and the periodic real-time task. In particular, the real-time task has been scheduled by a CBS with  $Q^s = 8.5ms$  and  $T^s = 20ms$ , which guarantees that all the task instances finish in a reservation period (within  $20ms$ ). The IRQ threads, instead, have been served using two (1, 10) CBSs. In this way, the maximum response time experienced by the periodic thread is controlled by the reservation period (and is always less than  $20ms$ ), and the network throughput is close to the maximum value (a throughput of  $79.7Mbps$  has been measured).

### 4 Additional Experiments

After verifying that reservation-based scheduling of the IRQ threads allows to find latency/throughput tradeoffs that are impossible with fixed priorities,

<sup>3</sup>If we care only about real-time tasks, and all of them are scheduled through appropriate reservations, then it is possible to use a work-conserving variant of the paradigm - see the “non hard” CBS algorithm.

some additional sets of experiments have been performed to test how the reservation's parameters  $Q^s$  and  $T^s$  can affect the network performance<sup>4</sup>.

| $Q^s$        | $T^s$         | Throughput |
|--------------|---------------|------------|
| 940 $\mu s$  | 2000 $\mu s$  | 93.26Mbps  |
| 1410 $\mu s$ | 3000 $\mu s$  | 93.8Mbps   |
| 1880 $\mu s$ | 4000 $\mu s$  | 93.71Mbps  |
| 2820 $\mu s$ | 6000 $\mu s$  | 93.87Mbps  |
| 4700 $\mu s$ | 10000 $\mu s$ | 93.4Mbps   |
| 9400 $\mu s$ | 20000 $\mu s$ | 93.82Mbps  |

**TABLE 2:** Network throughput with different server parameters

For example, it has been verified that if the network IRQ threads are reserved enough CPU time, then the maximum network throughput can be achieved (note that a constraint on the reservation period  $T^s$  must also be respected... The correct dimensioning of the reservation's parameters depends on the number of interrupts that the network card can buffer [15]).

This set of experiments has been performed by first running `netperf` and measuring the fraction of CPU time consumed by the hard IRQ thread of the network card, which resulted to be 0.47. Then, the test has been repeated with the IRQ thread attached to a reservation ( $Q^s, T^s$ ) with  $Q^s/T^s = 0.47$ . The results are shown in Table 2 and show that the measured throughput is always near to the maximum possible.

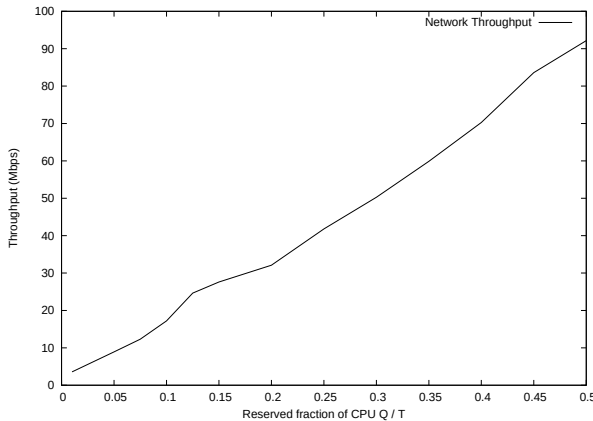


**FIGURE 3:** Network throughput as a function of  $Q^s/T^s$ .

By changing the ratio  $Q^s/T^s$  it is possible to control the network throughput, as verified in the next set

<sup>4</sup>Note that the network has been used as an example of additional hardware resource, but the presented techniques can be adapted to other devices.

of experiments, when the fraction of CPU time allocated to the network hard IRQ thread has been varied from 0.1 to 0.5. The results are displayed in Figure 3 and show that the achieved network throughput increases proportionally to  $Q^s/T^s$  until the reserved amount of CPU time is enough to properly serve the network driver.

| $T^s$ | $Q^s$<br>5ms | $Q^s$<br>10ms | $Q^s$<br>20ms | $Q^s$<br>40ms | $Q^s$<br>80ms |
|-------|--------------|---------------|---------------|---------------|---------------|
| 5ms   | 92,73        | 93,16         | 51,97         | 28,08         | 13,49         |
| 10ms  | -            | 93,15         | 93,6          | 51,69         | 27,07         |
| 20ms  | -            | -             | 93,44         | 92,75         | 51,53         |
| 40ms  | -            | -             | -             | 92,43         | 93,17         |
| 80ms  | -            | -             | -             | -             | 92,58         |

**TABLE 3:** Network throughput as a function of the reserved amount of time

Some similar results are shown in Table 3.

Finally, a last set of experiments has been performed to highlight the effects of the reservation period  $T^s$  on the network performance. Unless the reservation period is badly dimensioned causing dropped interrupts [15], it does not affect the network throughput (as shown, for example, in Table 3). However, it affects the network's responsiveness. This has been tested by adding a `ping` process in the experimental setup to measure the RTT (Round Trip Time) between the two test machines. The experimental results showed that increasing  $T^s$  can increase the maximum RTT (although the average value is not affected).

| $Q^s$ | $T^s$ | min<br>RTT | avg<br>RTT | max<br>RTT | mdev<br>RTT |
|-------|-------|------------|------------|------------|-------------|
| 1ms   | 3ms   | 0.062      | 0.109      | 16.498     | 0.289       |
| 2ms   | 6ms   | 0.057      | 0.105      | 36.504     | 0.368       |
| 3ms   | 9ms   | 0.058      | 0.103      | 38.684     | 0.379       |
| 4ms   | 12ms  | 0.058      | 0.101      | 50.991     | 0.428       |
| 5ms   | 15ms  | 0.059      | 0.102      | 50.928     | 0.453       |
| 6ms   | 18ms  | 0.058      | 0.103      | 52.814     | 0.507       |
| 7ms   | 21ms  | 0.059      | 0.104      | 79.782     | 0.566       |
| 8ms   | 24ms  | 0.057      | 0.104      | 92.568     | 0.634       |
| 9ms   | 27ms  | 0.057      | 0.106      | 86.688     | 0.685       |
| 10ms  | 30ms  | 0.062      | 0.106      | 152.257    | 0.783       |

**TABLE 4:** Round Trip Time (in ms) as a function of the reservation's parameters

Some of the results of this set of experiments are shown in Table 4: note that while the minimum and average RTT are not affected by the server period  $T^s$ ,

the maximum RTT increases almost linearly with it. As a consequence, the standard deviation of the RTT also increases with the server period. From these experiments it is possible to see that if small network latencies are needed then the network IRQ threads should be attached to reservations with a small  $T^s$ . On the other hand, a small server period can increase the number of context switches, so using larger values is preferred (when possible).

## 5 Conclusions

This paper presented some results obtained when using a reservation-based scheduler for serving the IRQ threads in a real-time version of Linux. The experiments performed with periodic real-time tasks and some applications sending and receiving data over the network showed that reservation-based scheduling of the IRQ threads allows to control both the latency experienced by real-time applications and the network throughput.

More experiments are of course needed, and future tests will be extended to other hardware devices such as disks, etc...

## References

- [1] *Cost and benefit of separate address spaces in real-time operating systems*, F. Mehnert, M. Hohmuth and H. Haertig, Dec 2002, Proceedings of the 23th IEEE Real-Time Systems Symposium
- [2] *Dresden Realtime Operating System*, R. Baumgartl, M. Borriß, H. Haertig, Cl.-J. Hamann, M. Hohmuth, L. Reuther, S. Schnberg and J. Wolter, Mar 1998, Proceedings of the Workshop of System-Designed Automation (SDA'98)
- [3] *Fitting Linux device drivers into an Analyzable Scheduling Framework*, T.P. Baker, A. Wang and M. J. Stanovich, Jul 2007, Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-time Applications
- [4] *Finding response times in a real-time system*, M. Joseph and P. Pandya, The Computer Journal, vol. 29, n. 5, 1986
- [5] *Modeling device driver effects in real-time schedulability analysis: Study of a network driver*, M. Lewandowski, M. Stanovich, T. Baker, K. Gopalan and Wang, 2007, Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium
- [6] *Cooperative Scheduling of Multiple Resources*, S. Saewong and R. Rajkumar, 1999, Proceedings of the IEEE Real-Time Systems Symposium
- [7] *Providing QoS by Scheduling Interrupt Threads*, G. Modena, L. Abeni and L. Palopoli, 2008, 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008) Work in Progress.
- [8] *Scheduling algorithms for multiprogramming in a hard real-time environment*, C. L. Liu and J. Layland, 1973, Journal of the ACM
- [9] *Internals of the RT Patch*, S. Rostedt, 2007, Proceedings of the Linux Symposium.
- [10] *Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems*, R. Rajkumar, K. Juvva, A. Molano and S. Oikawa, Jan 1998, Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking
- [11] *Integrating Multimedia Applications in Hard Real-Time Systems*, L. Abeni and G. Buttazzo, Dec 1998, Proceedings of the IEEE Real-Time Systems Symposium
- [12] *Implementing Resource Reservations in Linux*, L. Abeni and G. Lipari, Dec 2002, Proceedings of Fourth Real-Time Linux Workshop
- [13] <http://rt.wiki.kernel.org/index.php/Cyclictest>
- [14] <http://www.netperf.org>
- [15] *Resource Reservations and Device Drivers: Modelling and Implementation*, N. Manica, L. Palopoli, and L. Abeni. <http://www.disi.unitn.it/~abenipubs/irqrsv.pdf>