An EDF scheduling class for the Linux kernel *

Dario Faggioli, Fabio Checconi
Scuola Superiore Sant'Anna
Pisa, Italy
{d.faggioli, f.checconi}@sssup.it

Michael Trimarchi, Claudio Scordino
Evidence Srl
Pisa, Italy
{michael, claudio}@evidence.eu.com

Abstract

The Linux kernel is mainly used is general-purpose operating system, i.e., in server and/or desktop environments. During the last years, however, academic institutions and companies showed an increasing interest in using it for real-time and control applications as well.

However, since Linux has not been designed to be a real-time operating, the best-effort scheduling policy is not suited to provide high utilization and strong guarantees to time-sensitive tasks.

We present an enhancement of the Linux scheduler through the implementation of the well known Earliest Deadline First algorithm for real-time tasks, leaving the current behavior of existing policies unchanged. It is integrated with the latest Linux scheduler, support multicore platforms, it is available for embedded architectures (like ARM) and can be used with either periodic or aperiodic workloads.

1 Introduction

Linux is a General Purpose Operating System (GPOS) originally designed to be used in server or desktop environments. Since then, Linux has evolved and grown to be used in almost all computer areas. An important part of Linux is the process scheduler (or simply the *scheduler*). This component of the kernel selects which process to execute at any instant of time, and is responsible of dividing the finite resource of processor time between all runnable processes in the system.

During the last years, there has been a considerable interest in using Linux also for real-time and control, from both academic institutions and companies [15, 16]. Some reasons for this could be the free availability of its source code, the support for a great number of architectures, a rich set of already developed device drivers and the existence of billions of applications running on it.

Unfortunately, Linux has not been designed to be a Real-Time Operating System (RTOS), thus not much attention has been given to real-time issues. Therefore, making a classical real-time feasibility study of the system under development is not possible, and developers cannot be sure that timing requirements of tasks will be met under *every* circumstance. POSIX-compliant fixed-priority policies offered by Linux, on the other hand, are not much sophisticated and often do not suit the specific application requirements.
These issues are particularly critical when designing time-sensitive or control applications (e.g., MPEG players) for embedded devices like smart-phones. In

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

fact, when size, processing power, energy consumption and cost are tightly constrained, you need to efficiently exploit system resources, and at the same time meet the real-time requirements of the application.

It has to be said that companies exists that started selling modified versions of the Linux kernel with improved real-time support [17, 18, 19]. However, these non-standard versions of Linux are often non-free, and can not avail themselves of the support from the huge development community of the standard kernel.

Therefore, we believe that to be really "general", Linux should also provide enhanced real-time scheduling capabilities. In this paper, thus, we propose an implementation of the Earliest Deadline First (EDF) algorithm [1, 2], the well known real-time dynamic-priority scheduling algorithm.

The paper is organized as follows: at the beginning an overview of most notable real-time extensions for the Linux kernel proposed in the last decade is provided. Then, the current Linux scheduler is explained and the implementation of the proposed scheduling policy (`SCHED_EDF`) is provided. Last but not least, our implementation is evaluated and validated through a tests and experiments on real hardware, and finally, conclusions are driven.

2 Related work

During the last years, research institutions and independent developers have proposed several real-time extensions to the Linux kernel [4]. It must be said that, none of the extensions described in this section eventually became part of the official Linux kernel yet, thus making some of these projects obsolete.

First of all, in [20] we started investigating how an EDF policy could be implemented as a scheduling class of the new Linux modular scheduling framework, but we concentrated in uniprocessor systems, while the implementation presented here is multiprocessor. Moreover, the focus in [20] was on dealing with critical section, more than with scheduling itself.

Deadline based scheduling is also involved in [21], an even more recent work. However, what we do in there is to assign deadlines to fixed-priority task groups in the `sched_rt` class. Thus, the group to be scheduled is chosen by its deadline, while the task to be scheduled from within the group is selected according to its priority. On the other hand, in this work, a new scheduling class is utilized, and tasks have their own deadlines.

2.1 OCERA

A real-time scheduler for Linux 2.4 has been developed within the OCERA European project, and it is available as Open Source code [3, 7, 8]. To minimize the modifications to the kernel code, the real-time scheduler has been developed as a small patch and an external loadable kernel module. All the patch does is exporting toward the module (by some *hooks*) the relevant scheduling events.
The approach is straightforward and flexible, but the position where the *hooks* have to be placed is real challenge, and it made porting the code to next releases of the kernel very hard.

2.2 AQuoSA

The outcome of the OCERA project gave birth to the AQuoSA [23] software architecture. It basically consists on the porting of OCERA kernel approach to 2.6 kernel, with a user-level library for feedback based scheduling added.
Unfortunately, it lacks features like support for multicore platforms and integration with the latest modular scheduler [5, 6] and the cgroups filesystem.

2.3 Frescor

A real-time framework based on Linux 2.6 has been proposed by the Frescor European project [9]. It is based on AQuoSA and further adds to it a contract-based API and a complex middleware for specifying and managing the system performances, from the perspective of the Quality of Service (*QoS*) it provides.
Obviously, it suffers from all the above mentioned drawbacks as well.

2.4 LITMUS-RT

LITMUS-RT [10] is a plugin based scheduling framework with a wide variety of implemented real-time scheduling algorithms. The aim of the project, at least for now, is the evaluation of multiprocessor scheduling algorithms and synchronization protocols from research point of view.
Moreover, it only runs on Intel (x86-32) and Sparc64 architectures (i.e., no embedded platforms, the one typically used for industrial real-time and control).

2.5 RTLinux, RTAI, Xenomai

The so called *interrupt abstraction* approach [4] actually allows to schedule even *hard* real-time tasks on Linux. It consists of having an abstraction layer of virtual hardware below the Linux kernel, resulting in some kind of multithreaded RTOS with standard Linux and all its processes running as the lowest priority thread.
This approach has been successfully implemented in several existing frameworks, the most notable examples of which are RTLinux [12], RTAI [11] and Xenomai [13].
It is an efficient solution, as it allows to obtain very low latencies, but is also invasive, and, often, not all standard Linux facilities are available to tasks running with real-time privileges.

3 Modular Scheduling Framework

Recently (since release 2.6.23) the previous O(1) Linux scheduler has been replaced by a completely new modular scheduler implemented by Ingo Molnar [5, 6]. In this section we will try to point out some of its more important characteristics.

3.1 Scheduling classes

This "new" scheduler has been designed in such a way to introduce *scheduling classes*, an extensible set of scheduler modules. These modules encapsulate specific scheduling policies details that the core scheduler needs not to know. The binding between each policy and the related scheduler is done through a set of *hooks* (i.e., function pointers) provided by each scheduling class and called by the core scheduler.

Currently, Linux comes with two scheduling classes[1]:

- `sched_fair`: the *"Completely Fair Scheduler"* (CFS) algorithm, for `SCHED_NORMAL` and `SCHED_BATCH` policies. The idea here is to run tasks in parallel and at precise weighted speeds, in, so that each task receives a "fair" amount of processor share;
- `sched_rt`: the POSIX fixed-priority real-time scheduling, for `SCHED_FIFO` or `SCHED_RR` policies with 99 priority levels.

A (partial) list of the hooks a scheduling class may provide, by filling them in its own `struct sched_class` structure, follows:

- `enqueue_task(...)`: enqueues a task in the data structure used to keep all runnable tasks (runqueue); usually called when the task enters a runnable state;
- `dequeue_task(...)`: removes a task from the runqueue; usually called when the task stop being runnable;
- `yield_task(...)`: yields the processor for other tasks to have a chance to be run;
- `check_preempt_curr(...)`: checks if a task shall preempt the currently running task;
- `pick_next_task(...)`: chooses the most appropriate task eligible to be run next;
- `put_prev_task(...)`: makes a running task no longer running;
- `select_task_rq(...)`: chooses on which runqueue (i.e., on which CPU) a waking-up task has to be enqueued;
- `task_tick(...)`: accounts each periodic system tick to the running tasks.

3.2 Limits of current scheduling classes

Real-time tasks are *computational activities characterized, at least, by a worst case execution time and a timing constraint* [2]. These tasks must be executed by the RTOS in a correct order, so that each one completes within its timing constraints. A typical constraint is the *deadline*, i.e., is the instant the task execution is required to be completed, otherwise the results could turn out to be useless.

The default scheduling policies of Linux (i.e., `SCHED_NORMAL` and `SCHED_BATCH` cannot provide the guarantees a time-sensitive application may require. This is mainly because no concept of timing constraint (e.g. deadline) can be associated to a task in them. In fact, although it is possible to assign a share of the processor time to a task (or a group of tasks)[2], there is no way to specify that a task must execute for 20msec within 100msec, as it is possible with real-time scheduling algorithm, such as EDF. Moreover, the time elapsed between two consecutive executions of a task is not deterministic and can not

[1] actually, a special idle scheduling class also exists to implement the idle task

[2] CFS act according to a global, fixed period

be bound, since it highly depends on the number of tasks running in the system at that time.

Even POSIX-compliant fixed-priority policies (i.e., SCHED_RR and SCHED_FIFO) diverges from what the real-time research community refer to as "real-time" [2, 14]. For instance, as best-effort policies, they do not allow to assign timing constraints to tasks. Moreover, it is well known [1] that they provide lower performances in term of both schedulability guarantees [?] and flexibility (e.g., graceful degradation and bounded tardiness), both on uniprocessor and multiprocessor systems.

4 SCHED_EDF: features and implementation

In this section we present our new scheduling class, sched_edf, implementing the SCHED_EDF scheduling policy. The purpose of this paper is to give as much details as possible about this first implementation of the new policy.
As we will point out, thorough experimental evaluation to validate, and maybe correct, some of the design choices we made, is the subject of ongoing and future work by us.

All the information about the code, together with links to installation and usage instructions can be found at http://www.evidence.eu.com/sched_edf/

4.1 Main features

Our scheduling class implements the Earliest Deadline First (EDF [?]) algorithm and uses the Constant Bandwidth Server (CBS [22]) to provide bandwidth isolation among tasks. They both are well known and discussed in the real-time community, thus we are not going into too much details of them here, for space reasons.

This scheduling class has been developed from scratch, without starting from any existing project, taking advantage of the modularity currently offered by the Linux scheduler.

The implementation is aligned with the current mainstream kernel, and it relies on standard Linux mechanisms (e.g., control groups) to natively support multicore platforms and to provide hierarchical scheduling through a standard API.

4.2 Interaction with existing policies

The addition of our scheduling class to the Linux kernel does not change the behavior of the existing scheduling policies, neither best-effort and real-time ones, as shown in Figure 1).

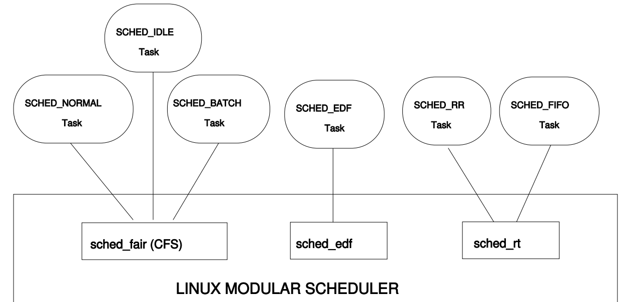

FIGURE 1: *Linux scheduler with SCHED_EDF.*

However, given the current Linux scheduler architecture, there is some interaction between scheduling classes. In fact, since each class is asked to provide a runnable task in the order they are chained in a linked list, "lower" classes actually run in the idle time of "upper" classes.

We decide to put our new scheduling class, sched_edf, in between the best-effort scheduling class, sched_fair, and the POSIX real-time scheduling class sched_rt. This means a ready EDF task always "win" the struggle for the CPU against a best-effort task, but always "loose" it against a fixed-priority POSIX real-time one. This has been done for the following reasons:

- backward application compatibility and (POSIX) standard compliance require SCHED_FIFO/RR tasks to run as soon as they can;
- it may be useful, either for the user or for the kernel (e.g. during system power down or CPU offlining), to have an easy way to specify an "always winner" task, without the need of dealing with the deadlines that are present in the system at that particular instant;
- since EDF is more efficient than fixed-priority in term of schedulability, it will be easier for it to achieve high utilization even if it runs in the idle time of fixed-priority, than the vice-versa;
- real-time analysis techniques already exists to deal with exactly such scheduling architecture.

It should be easy to notice that this solution has the drawback that EDF tasks will suffer from the

interference of fixed-priority tasks, but it has to be said that:

- the amount of interference coming from the user's `SCHED_FIFO/RR` tasks can be taken into account given the techniques cited before;
- the amount of interference coming from Linux system `SCHED_FIFO/RR` tasks can be accounted for as system overhead;
- in recent kernels, the bandwidth the whole `sched_rt` scheduling class occupies, can be forced to stay below a certain limit [3].

Experimental evaluation of the overhead and the experienced latency under different load conditions, the actual system schedulability and predictability, etc., is deferred to future works.

4.3 Multiprocessor scheduling support

In the real-time and scheduling community global and partitioned multiprocessor scheduling schemes exists. In global scheduling each task may be picked up and run on each CPU; in partitioned scheduling a task has its own CPU where it is always scheduled. Discussing advantages and drawbacks of both, is not in scope here, let us just say that the global case is usually thought and implemented by means of one global ready queue for all the CPUs, partitioning by one ready queue per CPU.

The way Linux deals with multiprocessor scheduling is often called *distributed runqueue*, which means each CPU as its own ready queue, as in partitioning; however, tasks can, if wanted or needed, migrate between the different queues. This adds the overhead of migrations but, at least, make it easy to pin some task on some processor (*scheduling affinity*) and, more important, addresses the serious locking scalability issues that –it is easy to figure out– concerns the single-runqueue solutions.

We are interested, for our new scheduling class, in a general solution, i.e. a globally scheduled system in which it is easy to ask one or more tasks to stay on a pre-specified CPU (or set of CPU). Therefore, the easiest and simplest way of achieving this is to go for the same approach of Linux itself, which means:

- we have one runqueue for each CPU, implemented with a *red-black tree*, for efficient manipulation;
- we migrate the tasks among the runqueues in a way such that:
  - we always try to have, on an m CPU system, the m earliest deadline ready tasks running on the CPUs;
  - we always respect the affinity the tasks specify.

[3] which is something we are trying to improve as well [21]

Therefore, if the affinity of a particular task is equal to only one precise CPU, that task will never be migrated and, when runnable, it always run only on it. Thus, it is sufficient that each EDF task in the system has its affinity correctly set, to achieve what we called partitioned scheduling, with very few overhead.

4.4 Tasks scheduling

The `sched_edf` scheduling class does not make any restrictive assumption on the characteristics of its task, thus, it can handle:

- periodic tasks, typical in real-time and control applications;
- sporadic tasks, typical in soft real-time and multimedia applications;
- aperiodic tasks.

A key feature of task scheduling in this scheduling class is that *temporal isolation* is ensured. This means, the temporal behavior of each task (i.e., its ability to meet its deadlines) is not affected by the behavior of any other task in the system. In other words, even if a task misbehaves, it is not able to exploit larger execution time than it has been allocated to it and monopolize the processor.

In fact, each task is assigned a *budget* (`sched_runtime`) and a *period*, considered equal to its *deadline* (`sched_period`). This means the task is guaranteed to execute for an amount of time equal to `sched_runtime` every `sched_period` (task *utilization* or *bandwidth*). When a task tries to execute more than its *budget* it is slowed down, by stopping it until the time instant of its next deadline. When, at that time, it is made runnable again, its budget is refilled and a new deadline computed for him. This is how the CBS [22] algorithm works, in its hard-reservation configuration.

However, although the CBS algorithm is very effective to encapsulate aperiodic or sporadic –real-time or non real-time– tasks in a real-time EDF

scheduled system, it imposes some overhead to "standard" periodic tasks. Therefore, we make it possible for periodic task to specify, before going to sleep waiting for the next activation, the end of the current instance. This avoid them (provided they behave well!) being disturbed by the CBS.

4.5 API for tasks

The existing system call `sched_setscheduler()` has not been extended, because of the binary compatibility issues that modifying its `struct sched_param` parameter would have raised for existing applications. Therefore, another system call, called `sched_setscheduler_ex(...)` is implemented. It allows to assign or modify the scheduling parameters described above (i.e., `sched_runtime` and `sched_period`) for tasks running with `SCHED_EDF` policy. The system call has the following prototype:

```
#define SCHED_EDF 6
struct sched_param_ex {
int sched_priority; /* Backward compliance */
struct timespec sched_edf_period;
struct timespec sched_edf_runtime;
};

int sched_setscheduler_ex(pid_t pid, int policy,
struct sched_param2 *param);
```

For the sake of consistency, also `sched_setparam_ex(...)` and `sched_getparam_ex(...)` have been implemented.

On the other hand, we did not add any system call to allow a periodic task to specify the end of its current instance, and the semantic of `sched_yield(...)` for EDF tasks has been modified for that purpose.

4.6 API for task groups

The scheduling class has also been integrated with the control groups mechanism in order to allow the creation of groups of tasks with a cap on their total utilization.

Therefore, the cgroups interface now provides two new entries, `cpu.edf_runtime_us` and `cpu.edf_period_us`. These files are created once the cgroup filesystem is mounted, to get and set the group bandwidth. In fact, a group has not actual budget or period/deadline, and these values are used to derive the utilization of the group. Consistency check is run when:

- a task is created or moved inside a group,
- the parameters of a task (if inside a group) are modified,
- a group is created or moved inside another group,
- the parameters of a group (if inside another group) are modified,

to check if the cumulative utilization of tasks and groups is below the one of the group that contains them (i.e., their *parent* group).

On multicore platforms, tasks and task groups can be moved among different processors using existing Linux mechanisms, i.e., `sched_setaffinity(...)` for tasks (either threads or processes), and `cpuset.cpus` for task groups.

5 Evaluation

Given the preliminary status of this work we report here only a very simple example showing our new policy at work.

First of all we modified the `schedtool` package to make it use the new `sched_setscheduler_ex(...)` system call, and thus be able to set `SCHED_EDF` as the scheduling policy of a task, with proper *budget* and *period*. We then ran a simple `yes` Linux process as a `SCHED_EDF` task with 100ms `sched_period` and 20ms `sched_runtime`. The actual schedule is obtained by using the `sched_switch` tracer from the `ftrace` infrastructure, recently introduced into the kernel. Finally, the output has been converted to VCD format by means of the `analyze` [4] program. Thus, we can use `GtkWave` to visualize what happens in the system. Notice that all the tools and programs mentioned above, are freely available and easy to install and use.

We can see what happens on the CPU where our `SCHED_EDF yes` process is being run in Figure 2).

Green "low" lines correspond to task execution, while yellow "high" ones to task being not runnable. Red vertical lines happens when a task become ready to run, and thus dark red rectangle means the task is ready but not running, usually because some other task of higher priority or scheduling class is keeping it out from the CPU. The `yes` EDF task is at the second line in the Figure; other lines represent other tasks running on the same CPU, as shown in the `Signals` box on the left. All these tasks but

[4] http://www.osadl.org/Visualize-the-temporal-relationship-of-L.taks-visualizer.0.html

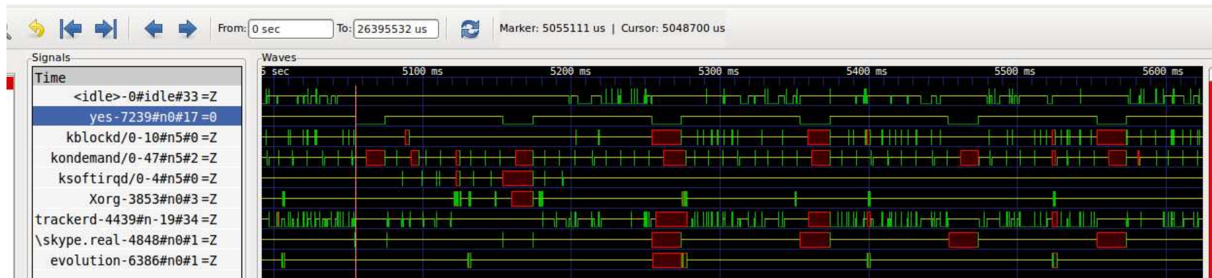

FIGURE 2: *Scheduling trace with a SCHED_EDF task*

`yes`, either being user or system tasks, run with `SCHED_NORMAL` as their scheduling policy.

Looking at the figure we notice at least the following:

- our `sched_edf` task runs as soon as it become runnable, preempting or preventing to be run all the best-effort tasks;
- thanks to the bandwidth isolation enforcing mechanism, even if it would like to run continuously, it is allowed to only exploit its full `sched_runtime` every `sched_period`, i.e., 20ms every 100ms.

6 Conclusions

In this paper, we presented an implementation of the Earliest Deadline First (EDF) algorithm for the Linux kernel.

With respect to similar work proposed in the past, our implementation takes advantage of the modularity offered by the new Linux scheduler, leaving the current behavior of existing policies unchanged. The new scheduling class is integrated with the latest Linux scheduler, and relies on standard Linux mechanisms (e.g., control groups) to natively support multicore platforms and to provide hierarchical scheduling through a standard API.

Our implementation does not make any restrictive assumption on the characteristics of the task. Thus, it can handle periodic, sporadic and aperiodic tasks. Another interesting feature of this scheduling class is the *temporal isolation* among the tasks handled. The temporal behavior of each task (i.e., its ability to meet its deadlines) is not affected by the behavior of the other tasks: if a task misbehaves and requires a large execution time, it cannot jeopardize the processor. Thanks to the temporal isolation property, each task executes as it were on a slower dedicated processor, so that it is possible to provide real-time guarantees on a per-task basis. Such property is particularly useful when mixing hard and soft real-time tasks on the same system.

References

[1] C.L.Liu, J.W.Layland, *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, Journal of the Association for Computing Machinery, 1973.

[2] John A. Stankovic, K.Ramamritham, M.Spuri, G.Buttazzo, *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*, Kluwer Academic Pubishers, 1998.

[3] L.Abeni, G.Lipari, *Implementing Resource Reservations in Linux*, Real Time Linux Workshop, 2002.

[4] G.Lipari, C.Scordino, *Linux and Real-Time: Current Approaches and Future Opportunities*, IEEE International Congress ANIPLA, 2006.

[5] *Design of the CFS scheduler*, Linux documentation, linux/Documentation/scheduler/sched-design-CFS.txt.

[6] I.Molnar, *Modular Scheduler Core and Completely Fair Scheduler*, http://lkml.org/lkml/2007/4/13/180

[7] I.Ripoll, P.Pisa, L.Abeni, P.Gai, A.Lanusse, S.Sergio, B.Privat, *WP1 — RTOS State of the Art Analysis: Deliverable D1.1 — RTOS Analysis*, OCERA, 2002.

[8] C.Scordino, G.Lipari, *A Resource Reservation Algorithm for Power-Aware Scheduling of Periodic and Aperiodic Real-Time Tasks*, IEEE Transactions on Computers, 2006.

[9] *FRESCOR: Framework for Real-time Embedded Systems based on COntRacts*, http://www.frescor.org

[10] J.Anderson and students, *LITMUS RT: Linux Testbed for Multiprocessor Scheduling in Real-Time Systems*, http://www.cs.unc.edu/ anderson/litmus-rt/

[11] L.Dozio, P.Mantegazza, *Real-Time Distributed Control Systems Using RTAI*, IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2003.

[12] V.Yodaiken, *The RTLinux Manifesto*, Fifth Linux Expo, 1999.

[13] P.Gerum, *The XENOMAI Project, Implementing a RTOS emulation framework on GNU/Linux*, 2002.

[14] G.Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Springer, 2005.

[15] G.Francis, L.Spacek, *Linux Robot with Omnidirectional Vision*, TAROS06, 2006.

[16] F.Kanehiro, K. Fujiwara, S.Kajita, K.Yokoi K.Kaneko, H.Hirukawa, Y.nakamura, K.Yamane, *Open Architecture Humanoid Robotics Platform*, IEEE International Conference on Robotics and Automation, Washington DC, 2002.

[17] Windriver Linux, http://www.windriver.com/products/linux/

[18] Timesys Embedded Linux kernel, https://linuxlink.timesys.com/3/Linux

[19] Montavista Real-Time Linux, http://www.mvista.com/real_time_linux.php

[20] D. Faggioli, M. Trimarchi, F. Checconi, M. Bertogna, A. Mancina, *An Implementation of the Earliest Deadline First Algorithm in Linux*, 24th ACM SAC, March 2009.

[21] Fabio Checconi, Tommaso Cucinotta, Dario Faggioli, Giuseppe Lipari, *Hierarchical Multiprocessor CPU Reservations for the Linux Kernel*, 5th OSPERT Workshop, July 2009.

[22] L. Abeni, G. Buttazzo, *Resource Reservation in Dynamic Real-Time Systems*, Real-Time Systems, Vol. 27, No. 2, pp. 123-167, July 2004.

[23] T. Cucinotta, L. Palopoli, L. Marzario, G. Lipari, *AQuoSA - Adaptive Quality of Service Architecture*, Software: Practice and Experience, April 2008, doi 10.1002/spe.883 Expand Abstract.