An implementation of the earliest deadline first algorithm in Linux

Conference Paper · January 2009

DOI: 10.1145/1529282.1529723 · Source: DBLP

3 authors:

Dario Faggioli
Scuola Superiore Sant'Anna
27 PUBLICATIONS **508** CITATIONS
SEE PROFILE

Michael Trimarchi
Scuola Normale Superiore di Pisa
7 PUBLICATIONS **190** CITATIONS
SEE PROFILE

Fabio Checconi
IBM
46 PUBLICATIONS **834** CITATIONS
SEE PROFILE

An Implementation of the Earliest Deadline First Algorithm in Linux [*]

Dario Faggioli
ReTiS Lab
Scuola Superiore Sant'Anna
Pisa, Italy
d.faggioli@sssup.it

Michael Trimarchi
ReTiS Lab
Scuola Superiore Sant'Anna
Pisa, Italy
trimarchimichael@yahoo.it

Fabio Checconi
ReTiS Lab
Scuola Superiore Sant'Anna
Pisa, Italy
fabio@gandalf.sssup.it

ABSTRACT

Recently, many projects have been started to introduce some real-time mechanisms into general purpose operating systems (GPOS) in order to make them capable of providing the users with some temporal guarantees. Many of these projects focused especially on Linux for its capillary and widespread adoption throughout many different research and industrial environments.

By tracking the kernel release cycle, we propose an efficient Earliest Deadline First implementation in the form of a patch-set against the 2.6.27 version, that is the latest released one, as of now. Our implementation provides the user with the possibility to choose SCHED_EDF as one of the possible scheduling policies for a task, with an enhanced version of the standard algorithm. In fact, we propose a new approach to shared resources' access which, differently from many other previous existing works, does not require the user to specify any parameters about the critical sections every task will enter during its execution.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management

Keywords

Linux, EDF, Shared Resources

1. INTRODUCTION

In the last years an increasing interest from the industrial world for real-time mechanisms within General Purpose Operating Systems (GPOS) has driven many development efforts towards introducing some modifications in the Linux Kernel [3] in order to provide timely guarantees to the otherwise completely unpredictable kernel-context execution.

Many projects exist in this area, tackling the problem from different perspectives trying to reach a common goal: to introduce the concept of deadline, that is an instant in time which the execution has to be carried out within. The vanilla kernel has no such notion, thus being not able to provide any kind of guarantees. The concept of real-time inside the kernel is limited to the Real-Time Extensions defined in the POSIX standard POSIX.1b (IEEE Std 1003.1b-1993) and not all of them are actually implemented (cfr. SCHED_SPORADIC scheduling policy). The real-time extensions describe some mechanisms based on the concept of priority, thanks to which it is possible to implement a preemptive fixed priority scheduling policy. This comes with some drawbacks, though. The biggest problem is about the maximum CPU bandwidth which it is possible to allocate using a fixed-priority scheme, which cannot reach the total CPU capacity. This leads to some wasted potential, whereas using a dynamic policy like the Earliest Deadline First [14], it would be possible to achieve a full CPU utilization.

In the last few Linux kernel releases, the scheduling subsystem has undergone a rather important re-engineering process, resulting in a completely modular structure giving the possibility to write new scheduling algorithms and plug them in the kernel as *scheduling classes*. Furthermore, a new standard scheduler, the Completely Fair Scheduler [5], has been proposed and implemented as one of the core scheduling modules. By using this module, the system is able to allocate the CPU to the requesting processes according to a fair share criterion. A fair allocation scheme, though, is usually not able to provide any kind of temporal guarantees, unless the weights are properly assigned at design time in order to meet the specific application requirements: this is not a viable approach, since the necessary computational effort increases dramatically.

We conceived and implemented a new scheduling class, implementing the EDF algorithm, making use of the new scheduling framework facilities. As such, we propose it in the form of a patch-set against the vanilla version of the kernel. We also conceived a new shared resources' access protocol which, differently from any standard protocols of this kind (as PCP [15] or SRP [11]), does not require an a-priori knowledge about which resources are accessed by which tasks and how long for, thus easing the role of the system designer.

[*] This work has been supported in part by the European Commission under the ACTORS (FP7/2008/ICT/216586) and PREDATOR (FP7/2008/ICT/216008) ICT projects.

SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.
Copyright 2009 ACM 978-1-60558-166-8/09/03 ...$5.00.

As it will be clear in the following sections we decided not to adhere to the standard algorithm definition as originally described by Liu and Layland [14] but to slightly modify it in order to provide some temporal isolation properties, which is a standard feature in reservation-based scheduling algorithms. We decided to enrich the algorithm with this property in order to face typical problems of soft real-time systems as temporal overruns.

The goal of our work is to provide a general purpose operating system like Linux with soft real-time guarantees, trying to minimize the number of deadline misses, even though a certain number of them can be tolerated without any dramatic consequences for the system schedulability analysis.

The remainder of the paper is organized as follows: in Section 2 we present the most important projects we want to compare against; in Section 3 we describe in detail the behaviour of the new modular scheduler core; in Section 4 we discuss our patch-set design and few implementation decisions with their advantages and drawbacks; in Section 5 we analyze some experiments trying to evaluate our code from different standpoints. Finally, in Section 6 we analyze our results and describe the future work.

2. RELATED WORK

As previously said, several projects exist in this area deserving at least a citation in this section. We could divide them into two categories according to different approaches to the problem.

2.1 Hypervisor layer

Probably, the three most important projects in this area are RT-Linux [8], RTAI [7] and Xenomai [10]. All of them share a common approach, that is to introduce a layer between the OS and the hardware, with the aim to totally separate the non-real time processes execution environment from the real-time one. This idea is enforced by intercepting all the interrupts: interrupts needed for deterministic computation are rerouted towards the real-time core, while other interrupts are forwarded to the non-real time operating system which runs at a higher level (and lower priority) than during working on real hardware.

We deem these approaches affected by the same drawback, that is a highly invasive approach with respect to the original OS. This may lead to higher difficulties in bugs scouting and fixing, as well as to more complex software platforms to maintain. In fact, the operating system core not running directly on the hardware may lead to the necessity of rewriting device drivers in order to make them aware of one more level of indirection (see, for example, the Real-Time Drivers Model [9]).

2.2 Kernel-level implementation

Red Hat and Timesys staff (Ingo Molnar, Thomas Gleixner et al.) have been carrying out a lot of work in this direction since 2004, providing the community with the RT-Tree [2], consisting of about 1.5 MByte of patches against the vanilla kernel (the latest release is for the 2.6.24 version) with the goal to make the Linux kernel behave according to hard real-time requirements. The patch-set consists of several improvements over the previous kernel versions, in particular in the following areas:

- in-kernel locking primitives become preemptive through the use of rt-mutexes which are also priority inheritance compliant;
- standard kernel spinlocks are now implemented through sleeping techniques making their protected critical sections preemptible;
- the former interrupt context is converted into preemptible kernel threads, giving the possibility to move the bottom half part of the handler code into the kernel thread;
- a high number of rescheduling points has been introduced in order to further reduce non-preemptible kernel code sections.

All these modifications are a significant step forward towards our goal, but there are still some issues to face. First of all, the kernel has no notion of time, when it comes to describe task properties, but the concept of time slice, that is the dynamically updated amount of time which a task may run for before the scheduler deallocates the CPU from it. This is the first point solved by our proposal.

Secondly, with the current state of the source code, only a fixed priority scheduling policy may be taken advantage of. As previously said, this can result in the CPU not being fully exploited, thus wasting system resources.

Finally, no multiprocessor scheduling issues are taken into account, leaving to the load balancing subsystem the distribution of processing threads within the system. Instead, we address this explicitly, by providing a multiprocessor partitioned scheduling algorithm implementation.

Another important piece of work we want to compare with is Litmus^{RT} [4]. It consists of a soft real-time extension to the Linux kernel in the form of a patch against the vanilla kernel (the current release relates to the 2.6.24 version). Its goal is to provide multiprocessor real-time scheduling and synchronization facilities. The Litmus kernel supports the sporadic task model with both partitioned and global scheduling approaches. We deem this work the most similar to our approach and more complete from the point of view of the possibilities the user has to configure the system scheduling part. As far as the shared resources access protocol is concerned, Litmus asks the user to specify the nature of the critical section a task must enter during its execution, choosing between a long and a short one. In contrast to this approach, we do not require a user to specify anything but a worst-case execution time and a period for each task, leaving as a system responsibility to manage the access to the shared resources and automatically recover from overload conditions.

3. LINUX SCHEDULING FRAMEWORK

Linux runs, except for `SCHED_SPORADIC`, a POSIX conforming scheduler with support for real-time and non real-time policies. Since Linux is a general purpose OS, the non real-time policy `SCHED_OTHER` is by far the most used. The code has quite recently been reworked, and turned into the so-called *Modular Scheduler Framework*, as well as provided with the group scheduling capability. Both these features are briefly described in the following subsections.

3.1 Modular Scheduler Framework

Since kernel release 2.6.23 "an extensible hierarchy of scheduler modules" is in place. Each scheduling module (*scheduling class*) is implemented in a different source file. Currently, there are only two modules: the fair scheduler module in `sched_fair.c`, for `SCHED_OTHER` tasks, and the real-time module (`sched_rt.c`), for `SCHED_FIFO` and `SCHED_RR` tasks.

The module hierarchy is made up by a linked list of available classes and the scheduler picks a ready task from the run-queue of the first module that has one. The interface each class has to implement is relatively small, i.e.:

- `enqueue_task()`
- `dequeue_task()`
- `requeue_task()`
- `task_tick()`
- `check_preempt_curr()`
- `pick_next_task()`
- `put_prev_task()`

Function names are self-explanatory, so we are not going into further details due to space reasons.

Both `sched_fair.c` and `sched_rt.c` provide the core scheduler (`sched.c`) with their own implementations of each of these functions. When the core scheduler calls them, the specific implementation of the scheduling class the current task belongs to gets invoked.

3.2 Linux and Group Scheduling

Group scheduling support has been recently introduced in the Linux kernel. This means that both tasks and tasks groups exist: they are considered as *scheduling entities*. Group scheduling entities have their own run-queues.

A more detailed description of this feature is out of the scope of this paper. We will extend our EDF implementation to support hierarchical scheduling in future works.

4. ALGORITHM DESCRIPTION

Our primary goal is to implement the standard EDF scheduling policy within the Linux Kernel, so that a real-time task may specify a minimum inter-arrival time and a worst case execution time. We assume implicit deadlines (equal to the periods). However, since Linux is a general purpose operating system, running also non real-time tasks, it is of paramount importance to correctly deal with overload conditions, as well as to face the problems of task blocking and critical sections accesses.

In case of overloads, i.e., a task trying to execute more than the WCET it specified, we force the task deactivation till the beginning of its next period, with a postponed deadline. As for critical section access arbitration the following sections will give details about the protocol we are proposing.

4.1 Critical Sections

One of the main problems in designing an efficient shared resource protocol is given by the difficulties in deriving tight upper bounds on the time spent in a critical section by a task. Since we do not want to charge the user with the task of providing such upper bounds, we developed an alternative strategy that is able to efficiently solve this problem. We chose to define a task parameter h_i specifying the maximal length for which a task τ_i can execute non-preemptively without missing any deadline. Inspired by the work developed by Baruah in [13], we provide here a simple way to compute a safe upper bound on the length of such non-preemptive chunks.

Assume tasks $\tau_1, \tau_2, \ldots, \tau_n$ are indexed in increasing deadline order, with $T_i \leq T_{i+1}$. Every task $\tau_k = (C_k, T_k) \in \tau$ is characterized by a worst-case computation time C_k, a period or minimum inter-arrival time T_k, and a relative deadline equal to the task period. The utilization of a task is defined as $U_k = \frac{C_k}{T_k}$. Let T_{min} be the minimum period among all tasks, and U_{tot} be the sum of the utilizations of all tasks.

THEOREM 1 $(O(n))$. *A task set that is schedulable with preemptive* EDF *remains schedulable if every task τ_k executes non-preemptively for at most h_k time units, where h_k is defined as follows*[1]*, with with $h_0 = \infty$*

$$h_k = \min\left\{h_{k-1}, \left(1 - \sum_{i=1}^{k} U_i\right) T_k\right\}. \tag{1}$$

PROOF. The proof is by contradiction. Assume a task set τ misses a deadline when scheduled with EDF, executing every task τ_k non-preemptively for at most h_k time-units, with h_k as defined by Equation (1). Let t_2 be the first missed deadline. Let t_1 be the latest time before t_2 in which there is no pending task with deadline $\leq t_2$. Consider interval $[t_1, t_2]$: since at start time no task is active, the interval is correctly defined, and the processor is never idled in $[t_1, t_2]$. Due to adopted policy, at most one job with deadline $> t_2$ can execute in the considered interval: this happens if such job is executing in non-preemptive mode at time t_1. Let τ_{np} be the task which such job, if any, belongs to. The demand of τ_{np} in $[t_1, t_2]$ is bounded by h_{np}. Moreover, $T_{\text{np}} > t_2 - t_1$. Every other task executing in $[t_1, t_2]$ has instead $T_i \leq (t_2 - t_1)$. Let τ_k be the task with largest period among these tasks. Then, $T_k \leq t_2 - t_1 < T_{\text{np}}$, and $h_k \geq h_{\text{np}}$.

Since there is a deadline miss, the total demand in interval $[t_1, t_2]$ must exceed the interval length:

$$h_{\text{np}} + \sum_{i=1}^{k} \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i > t_2 - t_1.$$

Using $x \geq \lfloor x \rfloor$ and $h_k \geq h_{\text{np}}$, we get

$$h_k + (t_2 - t_1) \sum_{i=1}^{k} U_i > t_2 - t_1, \tag{2}$$

$$h_k > (1 - \sum_{i=1}^{k} U_i)(t_2 - t_1). \tag{3}$$

And, since $t_2 - t_1 \geq T_k$,

$$h_k > (1 - \sum_{i=1}^{k} U_i) T_k,$$

reaching a contradiction. □

[1] The task with smallest relative deadline can execute non-preemptively during its whole WCET. The expression $(T_1 - C_1)$ is used to simplify the recursive formulation.

In order to avoid complex protocols to arbitrate the access to shared resources, a good programming practice is to keep the length of every critical section short [16]. If this is the case, preemptions can be disabled while a task is holding a lock, without incurring in significant schedulability penalties. Using Theorem 1, it is possible to derive upper bounds on the time for which each task may safely execute a critical section disabling preemptions. In Section 4.2 we will explain how to efficiently use such bounds.

An efficient implementation of the test of Theorem 1 has linear complexity in the number of tasks. We hereafter present a simpler corollary that can be used to derive weaker values for the available non-preemptive chunk length, with a reduced complexity.

COROLLARY 1 $(O(1))$. *A task set that is schedulable with preemptive* EDF *remains schedulable if* every task *executes non-preemptively for at most*

$$h = (1 - U_{tot})T_{\min}$$

time units.

The above theorem allows computing one single system-wide value h for the allowed maximum non-preemptive chunk length *of all tasks*, in a constant time. This lighter tests is a valid option whenever it is important to limit the overhead imposed on the system.

In the following sections, we will compare the solutions given by Theorem 1 and Corollary 1, in terms of schedulability performances and system overhead. Other more complex methods may be used to derive tighter values for the allowed lengths of non-preemptive chunks (see, for instance, [13]); nevertheless, we chose not to use such methods due to their larger (pseudo-polynomial) complexity. Having a fast method to calculate a global value for h is, in our opinion, really important, as in a highly dynamical system, with thousands of tasks, as Linux can be, it allows our method to be used without significant overhead.

Using a global value for h simplifies the implementation and reduces the runtime overhead of the enforcing mechanism, that has not to keep track of the per-task values.

It is worth noting that more sophisticate shared resource protocols like the Stack Resource Policy (SRP) [12] are not so suitable for the target architecture, since they are based on the concept of ceiling of a resource. To properly compute such parameter, it would be necessary to know a priori which task will lock each resource and, in a real operating system, this is definitely not a viable approach from a system design point of view.

4.2 Admission Control

One of the key points of our approach is that there is no need for the user to specify a safe upper bound on the worst-case length of each critical section, something that is very problematic in non-trivial architectures. The system will use all the available bandwidth left by the admitted tasks to serve critical sections, automatically detecting the length of each executed critical section, by means of a dedicated timer. If some task holds a lock for more than the allowed non-preemptive chunk length, it means that some deadline may be missed, and the system is overloaded. In this case, some decision should be taken to reduce the system load. There are many possible heuristics that can be used to remove some task from the system to solve the overload condition, the choice of which depends on the particular application. For instance, the system may reject the most recently admitted tasks; or it can reject tasks with heavier utilizations or longer critical sections, leaving enough bandwidth for the admission of lighter tasks; it can penalize less critical tasks, if such information is available; or it can simply ask the user what to do. We chose to reject the task with the largest critical section length, which is the one that triggered such scheduling decision executing for more than the allowed non-preemptive chunk length.

The system keeps track of the largest critical section R_i for each task τ_i, triggering a timer at the beginning and at the end of each critical section. Since every task will execute non-preemptively while holding a lock, one single timer is sufficient.

The admission control algorithm changes depending on the complexity of the adopted method to compute the time for which a task may execute with preemptions disabled. We distinguish into two cases: (i) using for all tasks a single system-wide value h given by Corollary 1; or (ii)using for each task τ_i a different value h_i given by Theorem 1.

In the first case the system keeps track of the largest critical section among all tasks: $R_{\max} = \max_{i=1}^{n}\{R_i\}$. For all deadlines to be met, this value should always be lower than the current non-preemptive chunk length:

$$R_{\max} \leq h. \tag{4}$$

When a task τ_k would like to be admitted into the system, the following operations are performed:

- The allowed non-preemptive chunk length h' after the insertion of the new task is computed using Corollary 1.
- If such value h' is lower than the maximum critical section length among the already admitted tasks $R_{\max}$, the candidate task is rejected, since it means that there would not be enough space available to allocate the blocking time of some task.
- Otherwise, task τ_k is admitted into the system, updating h to h'. Note that $R_{\max}$ does not need to be updated, since there is no available estimation of the maximum critical section length of τ_k (initially, $R_k = 0$).

When a task τ_k leaves the system, the new (larger) value of h is computed and accordingly updated. Moreover, if $R_k = R_{\max}$, $R_{\max}$ may as well be updated (decreased).

In a certain sense, we can say that a task is *conditionally* admitted into the system, and it will remain so as long as it does not show any critical section that is longer than the maximum non-preemptive chunk length allowed, in which case the task is rejected from the system. Alternative strategies may instead trigger different scheduling decision when $R_{\max}$ exceeds h, for instance creating room for a task with a long critical section by rejecting different tasks.

The slightly more complex case in which different non-preemptive chunk values h_i are used for each task τ_i, we will instead proceed as follows. In order to guarantee that all deadlines be met, we will check that every task τ_i has a non-preemptive chunk length h_i sufficiently large to accommodate the maximum critical section of that task:

$$\forall i, \;\; R_i \leq h_i. \tag{5}$$

When a task τ_k would like to be admitted into the system, the following operations are performed:

- Using Theorem 1, we compute the allowed non preemptive chunk length h'_i after the insertion of the new task, for all tasks τ_i having a period at least as large as τ_k's: $T_i \geq T_k$.
- If there is at least one value h'_i that is lower than the maximum critical section length of the corresponding task τ_i — i.e., $h_i < R_i$ — the candidate task τ_k is rejected.
- Otherwise, τ_k is admitted into the system, updating each h_i to h'_i.

When a task τ_k leaves the system, we simply recompute the h_i values of the tasks with period greater than T_k.

Independently from the adopted strategy, the system will check if an invariant condition (given by Equation (4) or Equation (5) is maintained. When it is not, some decision should be taken to solve the overload condition.

4.3 Exported Interface

One of the largest issues we faced at design time was deciding what interface our scheduling algorithm should export.

We strive for something not too different from the existing Linux scheduler interfaces. Moreover, we want the user to be able to code both periodic and sporadic tasks, as well as both hard and soft real-time applications. Finally, we think it would be useful to implement an already existing and widely adopted interface, so to make real-time programmers as comfortable as possible with it.

For all these reasons, we looked at the Ada 2005 [1] programming language specification, since EDF dispatching is included, as briefly shown below.

4.3.1 Ada 2005 EDF Dispatching Interface

Ada 2005 EDF dispatching package [6]:

```
with Ada.Real_Time;
with Ada.Task_Identification;
package Ada.Dispatching.EDF is
          subtype Deadline is Ada.Real_Time.Time;
          Default_Deadline : constant Deadline :=
                    Ada.Real_Time.Time_Last;
procedure Set_Deadline(D : in Deadline;
          T : in ADA.Task_Identification :=
          Ada.Task_Identification.Current_Task);
procedure Delay_Until_And_Set_Deadline(
          Delay_Until_Time : in Ada.Real_Time.Time;
          Deadline_Offset :
                    in Ada.Real_Time.Time_Span);
function Get_Deadline(
          T: Ada.Task_Identification.Task_ID :=
          Ada.Task_Identification.Current_Task)
          return Deadline;
end Ada.Dispatching.EDF
```

It is obvious how to exploit this interface to program either periodic or sporadic task. In particular, a call to Delay_Until_And_Set_Deadline() delays the calling task until time Delay_Until_Time. When the task becomes runnable again it will have

$\mathit{deadline} = \mathit{Delay\_Until\_Time} + \mathit{Deadline\_Offset}$

4.3.2 Linux EDF Interface

The Linux scheduler interface is based on `sched_{set, get}scheduler` and `sched_{set, get}param` system calls, and on the `sched_setscheduler` data structure. Since we definitely want to avoid binary compatibility problems with legacy applications, we did not change neither the behaviour of these functions nor the size of the data structures involved.

Moreover, as stated before, we want something similar to the Ada 2005 interface, provided that some adaptation to the specific context(i.e., the Linux kernel) is unavoidable.

Hence, we used a new data structure for EDF scheduling parameters, `sched_param2` and we added four new system calls:

- `sched_{set, get}scheduler2`;
- `sched_{set,get}param2`.

The new `sched_param2` has room to accommodate the period and the maximum possible runtime. It also contains a `deadline` field, useful for reading the current (absolute) deadline of a task.
In particular, the new `sched_setscheduler2` system call, which takes a `sched_param2` as an argument, behaves as follows:

1. it sets the parameters passed as the new current ones for the calling task;
2. it makes the task sleep until the relative time specified in the `sched_param2` argument passed, as the period;
3. on task wake-up, it sets its deadline to the current time plus the time value specified in the `sched_param2` argument that is passed, as the deadline.

Like in Ada 2005, both the sporadic and the truly periodic task models may be described through this interface.

4.4 Implementation details

Implementation has been carried out with efficiency in mind. The queue accommodating ready-to-run tasks is kept deadline-ordered, for $O(1)$ extraction. Furthermore, we implemented it as a red-black binary tree (RB-Tree) to keep also insertion and deletion efficient with a $O(logn)$ time complexity and because the kernel exports an highly optimized RB-Tree implementation, being it used in several other subsystems.

4.4.1 EDF Scheduling Class Implementation

Exploiting the modularity provided by the new scheduling framework, we implemented the EDF algorithm inside a new scheduling class. This means we added the file `sched_edf.c` and placed it as the head of the scheduling classes linked list, so that a ready EDF task will always have the highest priority in the system.

5. EXPERIMENTAL EVALUATIONS

To evaluate the effectiveness of our solution, we performed a series of experiments with randomly generated task sets.

We generated a great number of task sets, varying the total utilization U_{max} the task parameters T_k, C_k, and the length of the critical sections shared among the tasks. We used values of the total utilization U_{max} ranging from 0.4 to 0.9, and for each value, we evaluated different task behaviors.

We first considered values of U_k generated from an exponential distribution with $\lambda = 0.1$, with periods T_k uniformly distributed in the interval $[1, 1000]$; then we varied both λ and the range used for T_k generation. The acceptance rates for tasksets with $\lambda = 0.01$ and T_k generated from $[1, 10000]$ are shown in Figure 1; as expected, increasing the total utilization, the $O(1)$ test becomes less effective, while the $O(n)$ test shows a graceful degradation.

From the experiments, with the generated tasksets, we have observed that the values for h_k given by the $O(n)$ test were pretty uniform, indicating that the component given by the highest priority task often dominates the others. This is obviously positive, as it means that tasks are allowed longer non-preemptive regions, and can be seen as the increase of T_k dominating the decrease in U_k in the $O(n)$ formulation. The h_k values' span observed in this first test is shown in Figure 2. The figure shows max $\left(\frac{\max_k h_k - \min_k h_k}{T_{max}}\right)$ over all the generated tasksets of a given utilization.

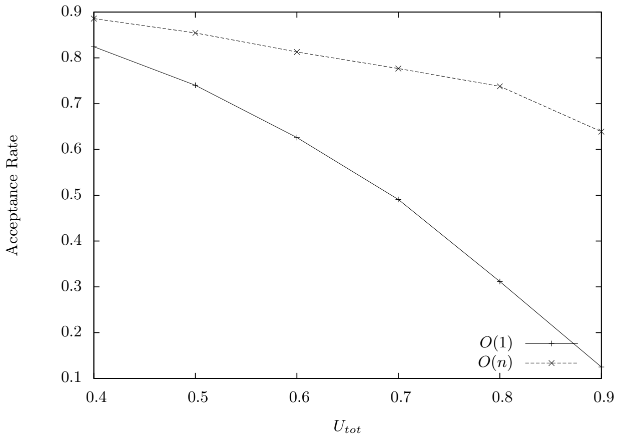

Figure 1: Light Tasks, Big Period Span.

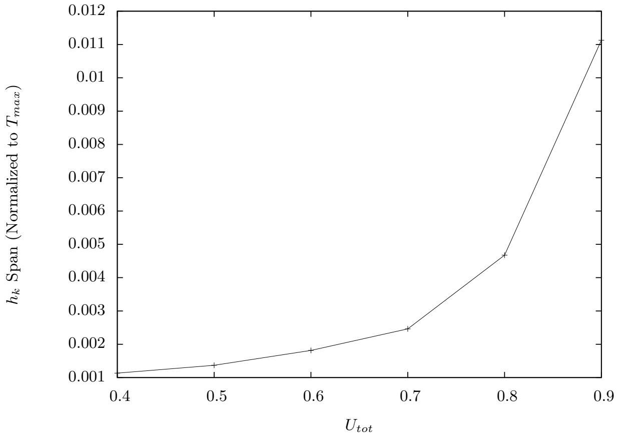

Figure 2: h_k Span for Light Tasks, Big Period Span.

6. CONCLUSIONS AND FUTURE WORK

In this work, we show how it is possible to modify the Linux kernel source code in order to cope with classical scheduling algorithms, such as EDF, obtaining a soft real-time enabled general purpose operating system.

With respect to prior work in this field, we attempted to better integrate the real-time scheduling mechanisms into the existing code. This has been achieved both with careful coding and introducing a Linux-friendly mechanism to deal with shared resources. Our shared resource access protocol does not require a-priori knowledge of the resource access patterns of the tasks and can be implemented efficiently: two key requirements in a dynamic system like Linux. We present also two fast methods to admit tasks in the system offering a different precision vs efficiency trade-off, part of an adaptation mechanism that regulates the load in the system on the basis of the behavior tasks are showing.

Currently, we are developing a new version of our patch which makes use of the group scheduling capabilities (described in Subsection 3.2) of the standard scheduler in order to obtain a hierarchical system. As such, we will be able to represent each scheduling entity as indistinguishable from the scheduling standpoint, whereas it might actually be one single task, as well as a group of them. By doing so, each level of the hierarchy could have a fraction of the bandwidth of the parent level, guaranteed by a centralized scheduler.

7. ADDITIONAL AUTHORS

Marko Bertogna, Antonio Mancina
Scuola Superiore Sant'Anna, Pisa - Italy,
{marko, a.mancina}@sssup.it

8. REFERENCES

[1] Ada 2005 Reference Manual. http://www.adaic.com/standards/ada05.html.
[2] Ingo Molnar's RT Tree. Available online.
[3] Linux kernel. The Linux Kernel Archives, http://kernel.org.
[4] Linux Testbed for Multiprocessor Scheduling in Real-Time Systems. http://www.cs.unc.edu/~anderson/litmus-rt/.
[5] Modular Scheduler Core and Completely Fair Scheduler. http://lkml.org/lkml/2007/4/13/180.
[6] Programming Real-Time with Ada 2005. http://www.embedded.com/showArticle.jhtml?articleID=192503587.
[7] RTAI home page. https://www.rtai.org/.
[8] RTLinux home page. http://www.rtlinux.org.
[9] The Real-Time Driver Model. http://www.xenomai.org/documentation/trunk/html/api/group__rtdm.html.
[10] XENOMAI home page. http://www.xenomai.org.
[11] T. P. Baker. A stack-based resource allocation policy for realtime processes. In *IEEE Real-Time Systems Symposium*, pages 191–200, 1990.
[12] T. P. Baker. Stack-based scheduling of real-time processes. *Real-Time Systems*, (3), 1991.
[13] S. Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, pages 137–144, Palma de Mallorca, Balearic Islands, Spain, July 2005. IEEE Computer Society Press.
[14] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.
[15] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.
[16] V. Yodaiken. Against priority inheritance. Technical report, Finite State Machine Labs, June 2002.