Dynamic Scheduling of Skippable Periodic Tasks in Weakly-Hard Real-Time Systems

Maryline Chetto
IRCCyN – University of Nantes
1 Rue de la Noé, F-44321 Nantes FRANCE
Maryline.chetto@univ-nantes.fr

Audrey Marchand
IRCCyN – University of Nantes
1 Rue de la Noé, F-44321 Nantes FRANCE
Audrey.marchand@univ-nantes.fr

Abstract

This paper deals with dynamic scheduling in real-time systems that have Quality of Service requirements. We assume that tasks are periodic and may miss their deadlines, occasionally, as defined by the so-called Skip-Over model. In this paper, we present a dynamic scheduling algorithm, called RLP (Red as Late as possible, a variant of Earliest Deadline to make slack stealing and get better performance in terms of ratio of periodic task instances which complete before their deadline. Simulation results show that RLP outperforms the two conventional skip-over algorithms, namely RTO and BWP, introduced about ten years ago. Finally, we present the integration of these QoS scheduling services into CLEOPATRE[1], a free open-source library which offers selectable real-time facilities on shelves.

1. Introduction

Real-time systems are computer systems in which the correctness of the system depends not only on the logical correctness of the computations performed, but also on time factors. Real-time systems can be classified in three categories: hard, soft and weakly-hard.

In hard real-time systems, all instances must be guaranteed to complete within their deadlines. In those critical control applications, missing a deadline may cause catastrophic consequences on the controlled system.

For soft real-time systems, it is acceptable to miss some of the deadlines occasionally. It is still valuable for the system to finish the task, even if it is late.

In weakly-hard real-time systems, tasks are allowed to miss some of their deadlines, but there is no associated value if they finish after the deadline. Typical illustrating examples of systems with weakly-hard real-time requirements are multimedia systems in which it is not necessary to meet all the task deadlines as long as the deadline violations are adequately spaced.

There have been some previous approaches to the specification and design of real-time systems that tolerate occasional losses of deadlines. Hamdaoui and Ramanathan in [1] introduced the idea of (m,k)-firm deadlines to model tasks that have to meet m deadlines every k consecutive invocations. The Skip-Over model was introduced by Koren and Shasha [2] with the notion of skip factor. It is a particular case of the (m,k)-firm model. They reduce the overload by skipping some task invocations, thus exploiting skips to increase the feasible periodic load.

In this paper, we address the problem of the dynamic scheduling of periodic task sets with skip constraints. In this context, the objective of a scheduling algorithm is to maximize the effective QoS (Quality of Service) of periodic tasks defined as the number of task instances which complete before their deadline.

The remainder of this paper is organized as follows: Section 2 presents relevant background materials about the Skip-Over model. We describe two basic scheduling algorithms, namely RTO and BWP which are based on this model. In section 3, we recall the foundation of EDL (Earliest Deadline as Late as possible) algorithm, a specific method to optimize system slack by running the hard deadline tasks at the latest time while still guaranteeing their timing requirements [3]. Then, we show how to use EDL for providing an efficient scheduling algorithm called RLP (Red as Late as Possible) for the Skip-Over model. Simulation results are reported in section 4 in order to show RLP performances compared

[1] work supported by the French research office, grant number 01 K 0742.

to RTO and BWP. In section 5, we provide an algorithmic description of the RLP scheduler. In section 6, we present the integration of these QoS scheduling services into CLEOPATRE, a free open-source library which offers selectable real-time facilities on shelves and we report measures in terms of footprint and time overheads. Section 7 summarizes our contribution and gives directions for future works.

2. Background materials

2.1. The skip-over model

In what follows, we consider the problem of scheduling periodic tasks which allow occasional deadline violations (i.e., skippable periodic tasks), on a uniprocessor system. We assume that tasks can be preempted at any time and they do not have precedence constraints. A task T_i is characterized by a worst-case computation time C_i, a period P_i, a relative deadline equal to its period, and a skip parameter s_i. This parameter represents the tolerance of this task to miss deadlines. That means that the distance between two consecutive skips must be at least s_i periods. When s_i equals to infinity, no skips are allowed and T_i is a hard periodic task. So, the skip parameter can be viewed as a QoS metric (the higher s_i, the better the quality of service).

Every task T_i is divided into instances where each instance occurs during a single period of the task. Every instance of a task is either red or blue [2]. A red task instance must complete before its deadline; A blue task instance can be aborted at any time. However, if a blue instance completes successfully, the next task instance is still blue.

2.2. RTO and BWP algorithms

Two scheduling algorithms were introduced about ten years ago by Koren and Shasha [2]. Under the *Red Tasks Only* (RTO) algorithm, red instances are scheduled as soon as possible according to *Earliest Deadline First* (EDF) algorithm [4], while blue ones are always rejected.

The *Blue When Possible* (BWP) algorithm is an improvement of RTO. Indeed, BWP schedules blue instances whenever their execution does not prevent the red ones from completing within their deadlines. In other words, blue instances are served in background relatively to red instances.

3. The RLP algorithm

3.1 Earliest Deadline as Late as possible

Let us review the fundamental properties of EDF algorithm, stated in [3] and [5] which are the basic foundation of our approach for scheduling tasks in the skip-over model. In general, implementation of EDF consists in executing tasks according to their urgency, as soon as possible with no inserted idle time. Such implementation is known as EDS (*Earliest Deadline as Soon as possible*).

Nevertheless, in some applications, this implementation presents drawbacks, for example when soft aperiodic tasks need to be served with minimal response times. In that case, it is preferable to postpone execution of periodic tasks, executing them by the so called EDL (*Earliest Deadline as Late as possible*) strategy. Such approach is known as Slack Stealing since it makes any spare processing time available as soon as possible. In doing so, it effectively steals slack from the hard deadline periodic tasks.

A means of determining the maximum amount of slack which may be stolen, without jeopardizing the hard timing constraints, is thus key to the operation of the EDL algorithm. In [3], we described how the slack available at any current time can be found. This is done by mapping out the processor schedule produced by EDL for the periodic tasks from the current time up to the end of the current hyper-period (the least common multiple of task periods). This schedule is constructed dynamically whenever necessary and is computed from a static EDL schedule which is constructed off-line and memorized by means of the two following vectors:

- K, called static deadline vector. K represents the time instants from 0 to the end of the first hyper-period, at which idle times occur and is constructed from the distinct deadlines of periodic tasks.
- D, called static idle time vector. D represents the lengths of the idle times which start at time instants of K.

The complexity for computing the EDL static schedule is O(N) where N is the total number of periodic instances in the hyperperiod.

At run time, the dynamic EDL schedule is updated from the static one by taking into account the execution of current ready tasks. It is described by means of the two following vectors:

- K_t, called dynamic deadline vector. K_t represents the time instants posterior to t in the current hyperperiod , at which idle times occur.
- D_t, called dynamic idle time vector. D_t represents the lengths of the idle times that start at time

instants given by K_t.

The complexity for computing the EDL dynamic schedule is O(K.n) where n is the number of periodic tasks, and K is equal to $\lfloor R/p \rfloor$, where R and p are respectively the longest deadline and the shortest period of current ready tasks [5].

3.2. Principles of RLP algorithm

The objective of RLP algorithm is to bring forward the execution of blue task instances so as to minimize the ratio of aborted blue instances, thus enhancing the QoS (i.e., the total number of task completions) of periodic tasks. From this perspective, RLP scheduling algorithm, which is a dynamic scheduling algorithm, is specified by the following behavior:

- if there are no blue task instances in the system, red task instances are scheduled as soon as possible according to the EDF (Earliest Deadline First) algorithm.
- if blue task instances are present in the system, they are scheduled as soon as possible according to the EDF algorithm (note that it could be according to any other heuristic), while red task instances are processed as late as possible according to the EDL algorithm. Deadline ties are always broken in favor of the task with the earliest release time.

The main idea of this approach is to take advantage of the slack of red periodic task instances. Determination of the latest start time for every red request of the periodic task set requires preliminary construction of the schedule as described previously and taking skips into account [6]. In the EDL schedule established at time t, we assume that the instance following immediately a blue instance which is part of the current periodic instance set at time t, is red. Indeed, none of the blue task instances is guaranteed to complete within its deadline.

Moreover, in [5] it was proved that the online computation of the slack time is required only at time instants corresponding to the arrival of a request while no other is already present on the machine. In our case, the EDL sequence is constructed not only when a blue task is released (and no other was already present) but also after a blue task completion if blue tasks remain in the system (the next task instance of the completed blue task has then to be considered as a blue one).

Note that blue tasks are executed in the idle times computed by EDL and are of same importance beside red tasks (contrary to BWP which always assigns higher priority to red tasks).

3.3. Illustrative example

To illustrate RLP, let us consider a set of five periodic tasks $T = \{T_0, T_1, T_2, T_3, T_4\}$ whose parameters are described in Table 1. We assume that all the tasks have the same skip parameter $s_i = 2$. We note that the processor utilization factor for this task set is equal to 1.15 and consequently some instances will necessarily miss their deadlines. It can be observed on Figure 1 that, thanks to RLP scheduling, the number of deadline violations relative to blue task instances has been reduced to three.

T_i	T_0	T_1	T_2	T_3	T_4
C_i	3	4	1	7	2
P_i	30	20	15	12	10

Table 1 : Task parameters

They occur at time instants t = 40 (task T_4), and t = 60 (tasks T_3 and T_4). Observe that T_3 first blue task instance which would fail to complete within its deadline with the BWP strategy, has enough time to succeed in the RLP schedule, since the execution of T_1 and T_0 first red task instances is postponed.

Until time t = 10, red task instances are scheduled as soon as possible. From time t = 10 to the end of the hyper-period (defined as the least common multiple of task periods), red task instances do execute as late as possible in the presence of blue task instances, thus enhancing the QoS of periodic tasks.

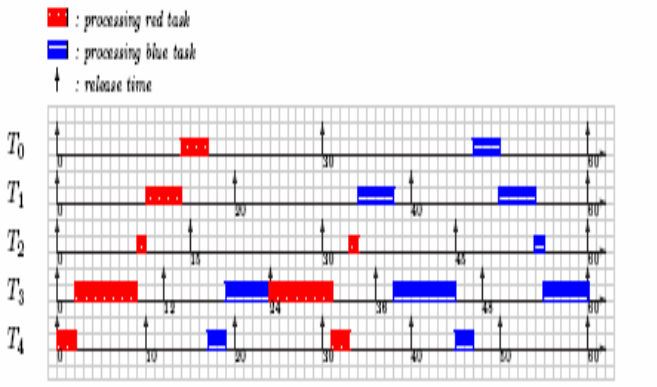

Figure 1 : A RLP schedule

4. Experimental results

4.1. Simulation parameters

The simulation context includes 50 periodic task sets, each consisting of 10 tasks with a least common multiple equal to 3360. Tasks are defined under QoS constraints with uniform s_i. Their worst-case execution time depends on the setting of the periodic load Up. Deadlines are equal to the periods and greater than or equal to the computation times. Simulations have been processed over 10 hyper-periods. Measurements rely on the ratio of periodic tasks instances which complete before their deadline. The evaluation is done by varying the periodic task load, Up.

4.2. Observations

Simulation results reported in Figure 2 and Figure 3 are carried out for a skip parameter s_i equal to 2 and 6 respectively, varying the periodic load and measuring the percentage of periodic task instances that complete successfully. We observe that, for any skip parameter and any processor workload, BWP and RLP outperform RTO for which the resulting QoS is constant and minimal. For $U_p \leq 1$, the processor is under-loaded, and both BWP and RLP success in completing all blue tasks instances which are respectively executed after and before red task instances. In overload situations, RLP reveals better than BWP and, higher is the skip parameter more significant is the advantage of RLP over BWP.

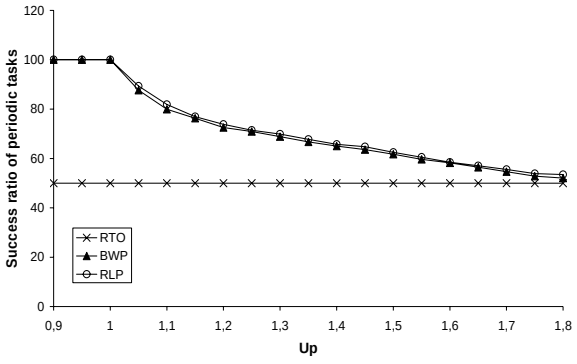

Figure 2 : QoS for uniform s_i=2

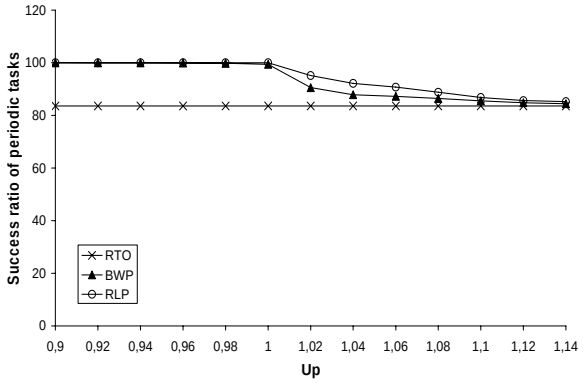

Figure 3 : QoS for uniform s_i=6

5. Implementing the RLP scheduler

The RLP scheduler is performed by the `RLP schedule()` function, reported hereafter. In our implementation, the scheduler maintains three task lists which are sorted in increasing order of deadline: waiting list, red ready list and blue ready list.

- waiting list: list of waiting tasks.
- red ready list: list of red scheduled tasks
- blue ready list: list of blue scheduled tasks

Note that tasks in the red ready list are always performed before any one present in the blue ready list . At `RLP schedule()` invocation time, the currently running task is the default candidate to run next.

```
RLP schedule(t : current time)
begin
/*Checking blue ready list in order to
abort tasks*/
while (task=next(blue ready list)=not(Ø))
if (task→release time+task→critical
delay<t) break
endif
Pull task from blue ready list
task→release time+=task→period
task→current skipvalue=1
Put task into waiting list
 Endwhile

 /*Checking waiting list in order to
release tasks*/
  while(task=next(waiting list)=not(Ø))
   if (task→release time>t) break
   endif
   if((task→current skipvalue<task→max
 skipvalue) and (Slack(t)=0))
     Pull task from waiting list
     Put task into red ready list
```

```
  else
     if (blue ready list=Ø)
        Compute EDL_schedule
     endif
     if (Slack(t)!=0)
        Pull task from waiting list
        Place task into blue ready list
     endif
  endif
 endwhile

/*Checking red ready list in order to
suspend tasks*/
 while(task=next(red readylist))=not(Ø))
  if (blue ready list=not(Ø)) and (
Slack(t)!=0)
    Pull task from red ready list
    Put task into waiting list
  endif
 endwhile
end
```

The `RLP schedule()` routine proceeds in three steps. In the first one, it examines blue ready list in order to or abort one or several blue tasks which have reached their deadline. The waiting list is scanned in the second step so as to resume tasks whose release time is less than or equal to current time. Red tasks are put in the red ready list when there is no slack at current time, contrary to blue ones released only when there is an idle time.

Slack value at time t is the output of the `Slack(t)` function, obtained from the EDL schedule. Such schedule is defined by computing the length of every processor idle time which follows every task deadline in the current hyper-period. In the last step, the red ready list is examined in order to suspend red ready tasks (released before current time), provided the blue ready list is not empty and there is slack at current time i.e. surplus processing time.

6. Integration in a free operating system

6.1. The Cleopatre library

A library of free software components was developed within the French National project CLEOPATRE (Software Open Components on the Shelf for Embedded Real-Time Applications) in order to provide more efficient and better service to real-time applications. Our purpose was to enrich the real-time facilities of real-time Linux versions, such as RTLinux [7] or RTAI [8]. RTAI was the solution adopted for this project because we wanted the CLEOPATRE components to be distributed under the LGPL license which is also the one used in the RTAI project.

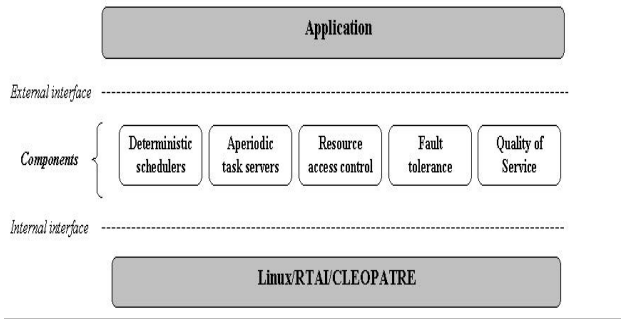

Figure 4 : The CLEOPATRE framework

The CLEOPATRE library offers selectable COTS (Commercial-Off-The-Shelf) components dedicated to dynamic scheduling, aperiodic task service, resource control access, fault-tolerance and now, QoS scheduling (see Figure 4). An additional layer named TCL (Task Control Layer) interfaces all the CLEOPATRE components. It has been added as a dynamic module in $RTAI DIR/modules/TCL.o, and represents an enhancement of the legacy RTAI scheduler defined in $RTAI DIR/modules/rt sched.o. CLEOPATRE applications are highly portable to any new CPU architecture thanks to this OS abstraction layer which makes the library of services, generic. The CLEOPATRE Off-the-Shelf components are optional except the OS abstraction layer (TCL) and the scheduler.

At most one component per shelf can be selected. Since all components of a given shelf have the same programming interface, they are interchangeable. Everything needed to use and develop CLEOPATRE can be downloaded from the web site of the project: http://cleopatre.rts-software.org.

RTO, BWP and RLP algorithms have been put into an additional shelf called Quality of Service. The QoS services are available as independent software components. This enables developers to build their own application-specific operating system.

6.2. Data structures and API

The basic data structure of our QoS schedulers is the task descriptor, defined in $RTAI DIR/include/QoS.h as `struct QoSTaskStruct`. This one contains nine fields for every task gathered and described in the following data structure:

```
typedef struct QoSTaskStruct
QoSTaskType;
struct QoSTaskStruct{
```

```
  void (*fct) (QoSTaskType *);
  /*pointer to task function*/
  TaskType TCL task;
 /*low-level descriptor*/
  TimeType critical delay; /*deadline*/
  TimeType period;
  TimeType release time;
  unsigned int max skipvalue;
  /*maximum tolerance to skips*/
  unsigned int current skipvalue;
  /*dynamic skip parameter*/
  unsigned int current shift;
 /*shift compared with a RTO sequence*/
  unsigned int slack;
  /* slack time of the task*/
  };
```

At initialization time, the user has to set the usual parameters for all tasks (period P_i, critical delay d_i,...) and also the additional skip parameter s_i for all QoS tasks.

The user interface for the QoS schedulers is composed of the following functions:

```
QoS create     : create a new task
QoS resume     : resume a task
QoS wait       : wait till next period
QoS delete     : delete a task
```

6.3. Overheads and footprints

For embedded real-time applications, the memory footprint and disk footprint of the operating system are generally key issues as well as the time overhead incurred by its execution. Measurements of footprints for the schedulers are given in Table 2.

Scheduler	Hard Disk size (Kb)	Memory size (Kb)
RTO	3.2	2.3
BWP	4.1	3.2
RLP	9.7	7.6

Table 2 : Footprints

We observe that RLP requires less than ten Kb. We have made some experiments to get a quantitative evaluation about the overhead introduced by the RLP scheduler. The tests have consisted in measuring the overhead for different number of tasks (5, 10, 15, 20,...) with all periods equal to 10 milliseconds. Periods of all tasks are harmonic, leading up to an hyper-period equal to 3360 ticks. Measurements were performed over a period of 1000 seconds on a computer system with a 400 MHz Pentium II processor with 384 Mo RAM. Results are shown in Figure 4.

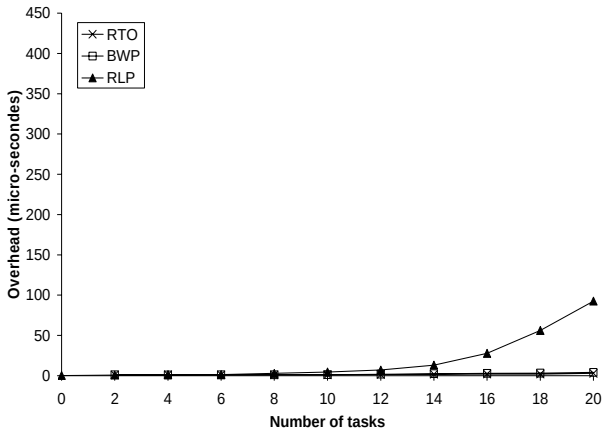

Figure 4 : Overheads of RTO, BWP and RLP

The timings shown hereafter were performed with a 1,7GHz Pentium 4 by using the Time Stamp Counter (TSC) available with every modern Intel processor. As it can been seen from Figure 4, the overhead of the QoS schedulers scales with the number of installed tasks. We note that BWP mean execution time is quite higher than the one observed for RTO. This is caused by the blue task management performed under BWP. The curve obtained for RLP is mainly due to the amount of time spent on the EDL schedule (performed only when a blue task instance is released or completed). As a matter of fact, we observe that overheads are closely related to algorithm efficiencies. An interesting feature of this component approach is that the selected scheduler can be tuned to balance performance versus complexity, so easily conforming to applications requirements.

6.4. A programming example

Success of real-time systems comes from both ease of use and performances. Writing code to run with CLEOPATRE is as simple as writing a C language program to run under Linux. The scheduling of QoS tasks is performed in the QoS schedule() function as described in §5. The scheduling occurs on timer handler activation (each 8254 interrupt).

Consider a periodic task set T composed of two tasks. The program implemented under Cleopatre is described below:

```
/*----Headers of all components—*/
#include <TCL.h>
#include <QoS.h>
#include <simul.h>

/*——————Timer clock period (10ms)———*/
#define TIMERTICKS 10000000

/*———————-Declaration of QoS tasks———————*/
QoSTaskType T1;
QoSTaskType T2;

/*——Code description of QoS tasks—-*/
void CodeT1() {simul.wait(4);}
void CodeT2() {simul.wait(1);}

/*———— initialization———————————*/
int init module(void)
{
TCLCreateType create={0, 2000, 0, 0};
/**********initializing QoS
tasks******/
QoS.create(&T1, CodeT1, 4, 20, 20, 2,
create);
QoS.create(&T2, CodeT2, 1, 15, 15, 2,
create);
QoS.resume(&T1,100);
QoS.resume(&T2,100);

/***starting the real-time mode***/
TCL.begin(TIMERTICKS, 20000);
return 0;}

/* ending and deleting QoS tasks */
void cleanup module(void)
{TCL.end();}
```

7. Concluding remarks and extensions

The paper has described scheduling algorithms dedicated to uni-processor systems that may experience overload. We have considered the Skip-over model where all tasks are periodic and characterized by a skip factor. Because using specific properties of Earliest Deadline, we have shown that the so-called RLP algorithm performs better than other skip-over strategies when the resulting QoS is measured in terms of global success ratio.

Recently, this approach was extended to cope with aperiodic tasks that arrive at unpredictable times [6]. Aperiodic tasks may have strict deadline or no deadline at all. The objective of the skip-over scheduler is then to serve all the tasks by accounting for both timing and QoS constraints.

In this paper, the order used by the scheduler for executing the blue instances is based on the relative deadline but could be selected based on a particular performance metric. We are no studying new strategies for achieving fairness while maximizing QoS [9]. We have compared different methods for ordering ready blue instances, aiming to balance individual success ratios for fairness motivations.

Finally, we have integrated these QoS functionalities in CLEOPATRE which is a portable, open-source, free to download and royalty free RTOS that can be used in commercial applications through the LPGL license.

References:

[1] M. Hamdaoui, P. Ramanathan, A Dynamic Priority Assignment Technique for Streams with (m,k)-firm deadlines. *IEEE Transactions on Computers*, Vol. 44, No. 4, pp 1443-1451, 1995.

[2] G. Koren., D. Shasha, Skip-Over Algorithms and Complexity for Overloaded Systems that Allow Skips. *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS'95)*, Pisa, Italy, 1995.

[3] H. Chetto, M.Chetto., Some Results of the Earliest Deadline Scheduling Algorithm. *IEEE Transactions on Software Engineering,* Vol. 15, No. 10, pp 1261-1269, 1989.

[4] C. Liu, J.W. Layland Scheduling algorithms for multiprogramming in a hard real-time environment .*Journal of ACM*, vol.20 n°1 pp.46-61, 1973.

[5] M. Silly-Chetto, The EDL Server for Scheduling Periodic and Soft Aperiodic Tasks with Resource Constraints, *Journal of Real-Time Systems*, Kluwer Academic Publishers, Vol. 17, pp 1-25, 1999.

[6] A. Marchand, M. Silly-Chetto,. Dynamic Real-Time Scheduling of Firm Periodic Tasks with Hard and Soft Aperiodic Tasks. *Journal of Real-Time Systems*. Vol. 32, No.1, pp 21-47, 2006.

[7] P. Mantegazza, DIAPM RTAI for Linux: Why's, what's and how's, Real Time Linux Workshop, University de Technology of Vienna, 1999.

[8]V. Yodaiken, the RTLinux Approach to Real-Time – FSM Labs Inc., August 2004.

[9] A.Marchand and M. Chetto, Stability and Robustsness Issues in Scheduling Periodic Tasks with Firm Real-Time Requirements, in Proc. Euromicro Conference on Real-Time Systems, WIP Session, Dresden, July 2006.