

# Dynamic Scheduling of Periodic Skippable Tasks in an Overloaded Real-Time System

Audrey Marchand\* and Maryline Chetto<sup>†</sup>, *member IEEE*

\*DISCA - Technical University of Valencia, Spain

<sup>†</sup>IRCCyN - University of Nantes, France

Email: {audrey.marchand, maryline.chetto}@univ-nantes.fr

## Abstract

*The need for supporting dynamic real-time environments where changes in workloads may occur requires a scheduling framework that explicitly addresses overload conditions, allows the system to achieve graceful degradation and supports a mechanism capable of determining the load to be shed from the system to handle the overload. In applications ranging from video reception to air-craft control, tasks enter periodically and have response time constraints, but missing a deadline is acceptable, provided most deadlines are met. Such tasks are said to be occasionally skippable and have an assigned skip parameter. We look at the problem of uniprocessor scheduling of skippable periodic tasks which consists in maximizing the robustness of the system defined as the global completion ratio. In this paper, we propose a novel scheduling Skip-over algorithm, called RLP/T, a variant of Earliest-Deadline First which adjusts the system workload such that tasks adhere to their timing and skip constraints and guarantees the best robustness.*

## 1. Introduction

While it is imperative that all time constraints are met in hard real-time systems, firm or soft real-time systems do not have as stringent timeliness requirements, allowing for some degree of tardiness (soft) or miss ratio (firm). A significant body of research within the soft and firm real-time area has focused on minimizing tardiness and/or miss ratio, but without quantifying acceptable levels. In this paper, we focus on scheduling firm periodic tasks which have additional requirements specifying the minimum acceptable completion ratios that should be met in order to maintain system correctness. In a hard real-time system all hard deadline tasks must meet their deadlines to maintain system correctness; otherwise, the system has failed. In contrast, a deadline is considered to be soft if it can be

missed occasionally (or not by much), and hence, a task missing a single soft deadline is not considered as a failure. In a soft real-time system correctness is determined by the degree to which timeliness has been enforced for the entire task set. However, completion of a tardy firm deadline task is not meaningful, since late delivery of the result is considered to be of no value to the real-time system. Although firm deadlines can occasionally be missed, there is normally an upper limit on the number of misses within a defined interval. The hard real-time paradigm is well established and it has received considerable attention by researchers and practitioners within academia and industry. Numerous techniques and algorithms, especially in the area of scheduling, have been developed. Most scheduling algorithms developed for soft and firm real-time systems lack the ability to enforce constraints on the upper limit of misses. Without such enforcement, unbounded consecutive time constraint violations may occur. Realistically, if consecutive instances of a task fail to complete before their deadlines, then the system will eventually suffer from a failure. This indicates that there are additional constraints expressing the minimum degree of timeliness for firm real-time tasks, that must be enforced. Unfortunately, little work has been done on scheduling of firm tasks with such constraints, and this is the subject of our paper. The most significant firm real-time constraint is  $(m, k)$ -firm constraint and consists of guaranteeing  $m$  out of  $k$  consecutive task executions or message transmissions. New strategies to schedule systems with  $(m, k)$ -firm constraints or other similar constraints (*e.g.* window-constrained [13] and skip-over [5]) have been defined. Koren and Shasha identified a model, called skip-over, for overloaded systems allowing skips. In this paper, we focus on dynamic scheduling of periodic tasks with firm deadlines, where each firm task has an additional constraint specifying the acceptable level of timeliness, the skip factor. If a task has a skip factor  $s$ , it will have at most one invocation skipped out of  $s$ . We have presented, in an earlier paper, a scheduling framework [9],

RLP (Red Tasks as Late as possible) for handling dynamic workloads consisting of periodic firm deadline tasks. In this paper, we introduce a variation of RLP, called RLP/T. We add a novel admission control mechanism to RLP, which enables the algorithm to enforce robustness constraints by performing a dynamic test to accept or not a skippable task for execution, so as to lost minimum processing time. The admission controller tests for schedulability of a new instance upon its arrival. It acts as a task filter guaranteeing that the scheduler is always able to find a feasible schedule. Our performance evaluation shows that RLP/T outperforms the basic skip-over algorithms, RTO (Red Tasks Only) and BWP (Blue When Possible) [5]. The paper is structured as follows. In section 2 we describe the Skip-Over model and recall some background materials in section 3. In section 4 and 5 we respectively describe in detail the algorithm RLP and its extension RLP/T. In section 6 we present results of a simulation study. Implementation and validation issues are considered in section 7. At the end of the paper, we make a short summary and directions for future work in section 8.

## 2. The skip-over model

### 2.1 Terminology and assumptions

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. The system under study has a workload consisting of tasks that 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.

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 [5]. A red task instance must complete before its deadline whereas 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. The first one proposed by Koren and Shasha is 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 second algorithm studied is the Blue When

Possible (BWP) algorithm which is an improvement of the first one. 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.

### 2.3 Illustration

Hereafter is presented a scheduling example with two tasks  $T_i(C_i, P_i, s_i)$  defined according to the Skip-Over model. Task  $T_1$  is a hard real-time periodic task ( $s_i = \infty$ ) while task  $T_2$  allows deadline skips ( $s_i = 3$ ). Tasks are scheduled as soon as possible according to their deadline.

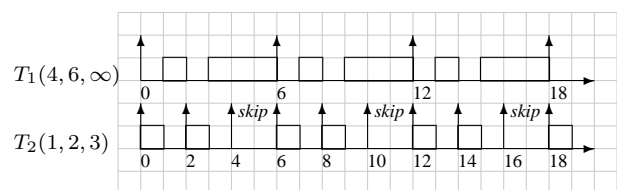


Figure 1. A Skip-Over schedule

The system is overloaded ( $U_p = \sum_{i=1}^n \frac{C_i}{P_i} = \frac{4}{6} + \frac{1}{2} = 1.17$ ), but we can see that tasks can be schedulable provided  $T_2$  exactly skips one instance every 3.

## 3. Background material on Earliest Deadline

Despite of not being implemented yet in any commercial RTOS, dynamic scheduling policies and more particularly Earliest Deadline First algorithm (EDF) offer many advantages such that accomplishing a 100% processor utilization. Let us review the fundamental properties of Earliest Deadline First algorithm, stated in [3] [5] which are the basic foundation of our approach for scheduling tasks in the skip-over model.

### 3.1. Earliest Deadline as Soon as possible (EDS)

In general, implementation of EDF consists in executing tasks according to their urgency, as soon as possible with no inserted idle times. 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 the execution of periodic tasks, executing them by the so-called EDL (Earliest Deadline as Late as possible) strategy.

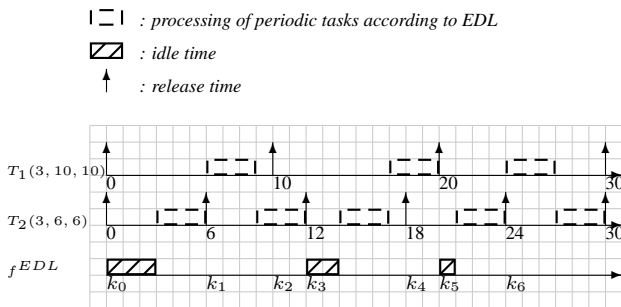
### 3.2. Earliest Deadline as Late as possible (EDL)

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 hyperperiod (the least common multiple of task periods). This schedule is constructed dynamically whenever necessary and is computed from a static EDL schedule, 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 hyperperiod, 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. Formula that give  $K$  and  $D$  can be found in [11].

The computation of the EDL static schedule is illustrated in Figure 2 with a set of two task  $T_i(C_i, P_i, D_i)$ . Note that in this example there are 7 potential idle times denoted by  $k_i, i = 0, \dots, 6$ . Results of the computation of vectors  $K$  and  $D$  are summarized in Table 1.



**Figure 2.** EDL computation of static idle times

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:

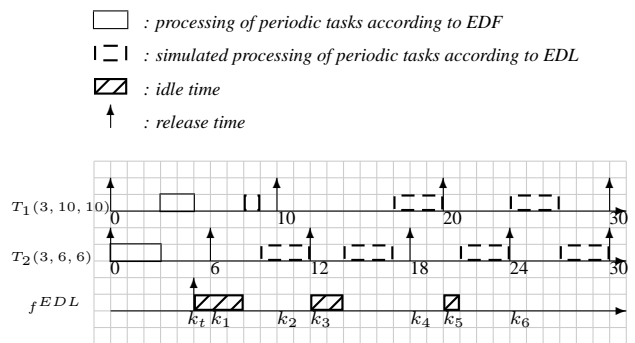
**Table 1.**  $K$  and  $D$  computation

$K$	0	6	10	12	18	20	24
$D$	3	0	0	2	0	1	0

- $K_t$ , 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  $\lfloor \frac{R}{p} \rfloor$ , where  $R$  and  $p$  are respectively the longest deadline and the shortest period of current ready tasks [11]. Formula that give  $K_t$  and  $D_t$  can be found in [11].

Assume now that, given the same task set as introduced before, we want to compute idle times from time instant  $t = 5$  when tasks are processed by EDS from 0 to  $t$ . The resulting schedule is depicted in Figure 3.



**Figure 3.** EDL computation of dynamic idle times at time  $t = 5$

**Table 2.**  $K$  and  $D$  computation

$K_t$	5	6	10	12	18	20	24
$D_t$	1	2	0	2	0	1	0

Then, from time  $t = 5$  until the end of the hyperperiod, tasks are scheduled as late as possible according to EDL. Nonzero idle times resulting from the computation of vectors  $K_t$  and  $D_t$  (see Table 2) appear at times  $t = 5$ ,  $t = 6$ ,  $t = 12$  and  $t = 20$ .

Chetto and Chetto in [3] showed that the EDL schedule computation can be efficiently used for improving the

service of aperiodic tasks. By definition, soft aperiodic requests must not compromise the guarantees given for periodic tasks and should be completed as soon as possible. No acceptance test is performed for soft aperiodic requests; they are served on a best-effort basis within the computed idle times, the goal being to minimize their response times. With respect to hard aperiodic tasks, every task is subject to an acceptance-rejection test upon arrival. Indeed, given their absolute deadline and their worst-case execution time, hard aperiodic tasks can easily be admitted or rejected on the basis of the knowledge of idle times localization.

In [4] the same authors presented how EDL can be used to generate fault-tolerant schedules. The authors assume that every task is composed of primary and alternate jobs as specified by the Deadline Mechanism model [2]. The main feature of their method lies in the ability of dynamically changing the schedule and accounting for runtime situations such as successes or failures of primaries.

In our approach, we propose to use the EDL scheduling algorithm with the Skip-Over task model ascertaining the fact that the Deadline Mechanism and the Skip-Over models present analogous features. In both cases, the objective is to manage overload situations. The Deadline Mechanism performs commutations to a degraded mode in which results of lower quality are produced, but still meeting all the deadlines. As regards the Skip-Over model, it also performs commutations to a degraded mode, processing results of constant quality, but allowing some results to be not produced at all. Blue task instances of the Skip-Over model can be compared to primary jobs of the Deadline Mechanism, whereas red task instances can be associated with alternate jobs.

## 4. The RLP algorithm

### 4.1. Principles

The objective of RLP algorithm [9] is to bring forward the execution of blue task instances so as to minimize the ratio of aborted blue instances, thus enhancing the robustness (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 behaviour:

- 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.

### 4.2. Applying EDL to Red instances

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 instance of the periodic task set requires preliminary construction of the schedule as described previously and taking skips into account [8, 10]. In the EDL schedule established at current 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 [11] it was proved that the online computation of the slack time is required only at time instants corresponding to the arrival of an instance 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).

### 4.3. Illustrative example

To illustrate RLP, let us consider a set of five periodic tasks  $\mathcal{T} = \{T_1, T_2, T_3, T_4, T_5\}$  whose parameters are described in Table 3. 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.

**Table 3.** Task parameters

$T_i$	$T_1$	$T_2$	$T_3$	$T_4$	$T_5$
$C_i$	3	4	1	7	2
$P_i$	30	20	15	12	10

It can be observed that, thanks to RLP scheduling, the number of violations of deadline relative to blue task instances equals three. They occur at time instants  $t = 40$  (task  $T_5$ ), and  $t = 60$  (tasks  $T_4$  and  $T_5$ ). Until time  $t = 10$ , red task instances run as soon as possible. From time  $t = 10$  to the end of the hyperperiod, red task instances do execute as late as possible in the presence of blue task instances, thus enhancing the robustness of periodic tasks.

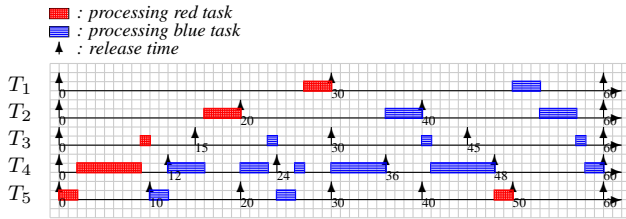


Figure 4. A RLP schedule ( $s_i = 2$ )

## 5. The RLP/T algorithm

### 5.1. Principles

The main disadvantage of RLP lies in that this scheduler attempts to process as soon as possible blue instances at the risk of not completing them before their deadlines due to possible overload. Such strategy then generates wasted processing times due to incomplete execution of tasks and does not provide the best degradation during overload periods. Consequently, we propose to handle overload by performing admission control before deciding to execute a ready blue instance. The so-called RLP/T (Red Tasks as Late as Possible with blue acceptance Test) then leads to maximize the robustness. Considering the worst-case execution time of each occurring blue instance we can ensure that the set of admitted blue instances is feasibly schedulable. And as soon as a blue instance is admitted, the next task instance of the completed blue instance which was necessarily red is considered as a blue one. A blue instance is normally rejected once it is determined that it cannot be admitted given the current workload composed of red instances and already admitted blue instances.

### 5.2. The admission test

The acceptance test of blue tasks within a system involving Skip-Over tasks presented below in Theorem 1, is based on results established by Silly and al. [12] for the acceptance of sporadic requests occurring in a system consisting of basic periodic tasks (*i.e.*, without skips).

**THEOREM 1** *Task  $B$  is accepted if and only if, for every task  $B_i \in \mathcal{B}(\tau) \cup \{B\}$  such that  $d_i \geq d$ , we have  $\delta_i(\tau) \geq 0$ , with  $\delta_i(\tau)$  defined as:*

$$\delta_i(\tau) = \Omega_{\mathcal{T}(\tau)}^{EDL}(\tau, d_i) - \sum_{j=1}^i c_j(\tau) \quad (1)$$

$\mathcal{B}(\tau)$  denotes the blue task set supported by the machine at time  $\tau$ .  $\delta_i(\tau)$  is called slack of task  $B_i$  at time  $\tau$  and represents the maximum units of time during which the task

could not be served by the processor without missing its deadline.  $\Omega_{\mathcal{T}(\tau)}^{EDL}(\tau, d_i)$  denotes the total units of time that the processor is idle in the time interval  $[\tau, d_i]$ . The total computation time required by blue tasks within  $[\tau, d_i]$  is given by  $\sum_{j=1}^i c_j(\tau)$ .

Hence, a blue task occurring at time  $\tau$  can be accepted provided that all the slacks (including the occurring task's one) of the blue tasks having a deadline greater than or equal to the occurring task, which are computed at  $\tau$ , remain greater than or equal to zero.

### 5.3. Example

RLP/T scheduling is illustrated in Figure 5 with the periodic task set  $\mathcal{T}$  defined in Table 3. It is easy to see that RLP/T improves on RLP. Only two violations of deadline relative to blue task instances are observed: at time instants  $t = 40$  (task  $T_5$ ) and  $t = 60$  (task  $T_4$ ). The acceptance test contributes to compensate for the time wasted in starting the execution of blue tasks which are not able to complete within their deadline. As we can observe, in the RLP case (see Figure 4),  $T_4$  blue instance released at time  $t = 48$  is aborted at time  $t = 60$  (2 units of time were indeed wasted). Note that the rejection of this blue task instance, performed with RLP/T, contributes to save time used for the successful completion of  $T_5$  blue instance released at time  $t = 50$ .

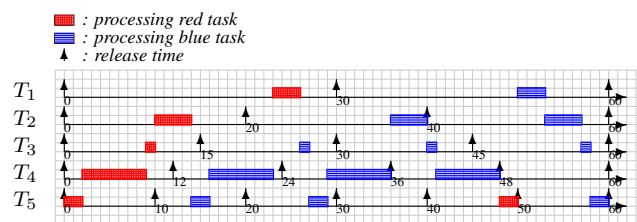


Figure 5. A RLP/T schedule ( $s_i = 2$ )

## 6. Performance evaluation

### 6.1. Simulation details

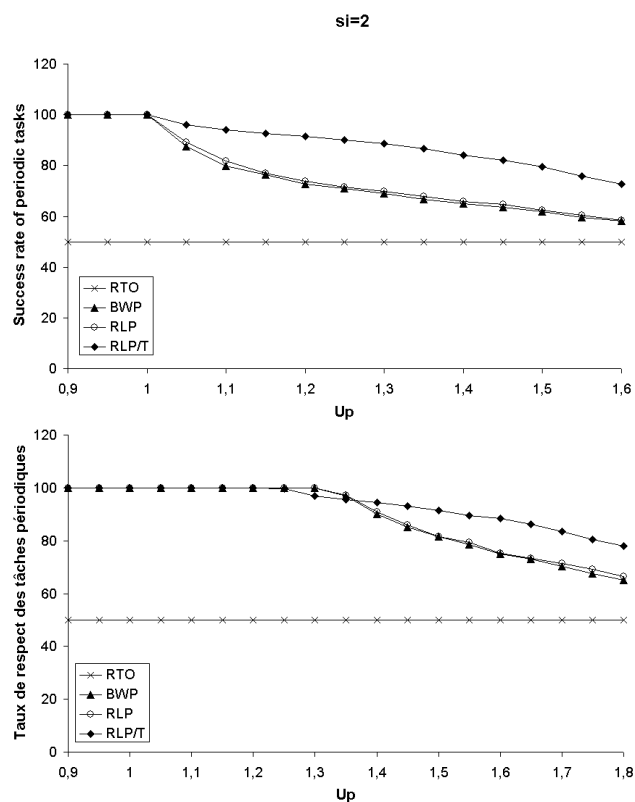
We report part of a performance analysis which consists of three simulation experiments designed to evaluate RLP/T with respect to RTO, BWP and RLP. In the first experiment, we measure the robustness, in the second one, the wasted time ratio (*i.e.* the percentage of unuseful processing time) and in the third one, the percentage of processor idle time. The study is done by varying the periodic task load  $U_p$ .

The simulator generates 50 periodic task sets. Each one contains 10 tasks with a least common multiple equal to

3360. Tasks are defined with uniform skip factor  $s_i$ . The worst-case computation times depend on the setting of the periodic load  $U_p$ . Deadlines are equal to the periods and greater than or equal to the computation times. Simulations have been processed over 10 hyperperiods.

## 6.2. Experiment 1

Simulation results reported in Figure 6 are carried out for a skip parameter  $s_i$  equal to 2, varying the periodic load and measuring the robustness given by the percentage of periodic task instances that complete successfully. We report here the results of 2 simulation studies where tasks have an actual computation time (ACET) equal to 100% and 75% of the worst-case computation time (WCET) respectively. Let us recall that in practice, tasks have variable actual computation times assumed to be less than an estimated worst-case computation time. The assumption that a task will consume its WCET in all the activations does not have to be necessarily true, which implies that the real utilization of the CPU is less than the estimated in the schedulability test.



**Figure 6.** Robustness for ACET=WCET and ACET=0.75\*WCET

On one hand, we observe that, for any processor workload, BWP and RLP outperform RTO for which the robust-

ness is constant and minimal. For  $U_p \leq 1$ , the processor is underloaded, and both BWP and RLP succeed in completing all blue task instances which are respectively executed after and before red task instances. In overload situations, RLP and BWP give quite the same performances. However, compared with RLP, RLP/T provides a significant performance improvement.

On the other hand we can see that, for an identical periodic load, the success ratio of tasks observed for BWP, RLP et RLP/T is higher when the task execution time is less than its worst-case execution time. This is due to the fact that the amount of time (WCET-ACET) that is not used by each instance and which is in fact additional CPU time, is used for completing a greater number of task instances.

Moreover, note that for low overloads and ACET=0.75\*WCET, BWP and RLP outperform RLP/T. This is due to the fact that the admission test performed by RLP/T is based upon the assumption that tasks execute according to their WCET, the value of ACET being not known *a priori*. Consequently, RLP/T will necessarily reject tasks that after all could have been accepted on the basis of their ACET. This is exactly what we can observe for the periodic load equal to 130%, RLP/T temporarily offers lower performances than BWP and RLP. Note that this phenomenon is not observable any more once the skip parameters are higher (e.g.  $s_i = 6$ ).

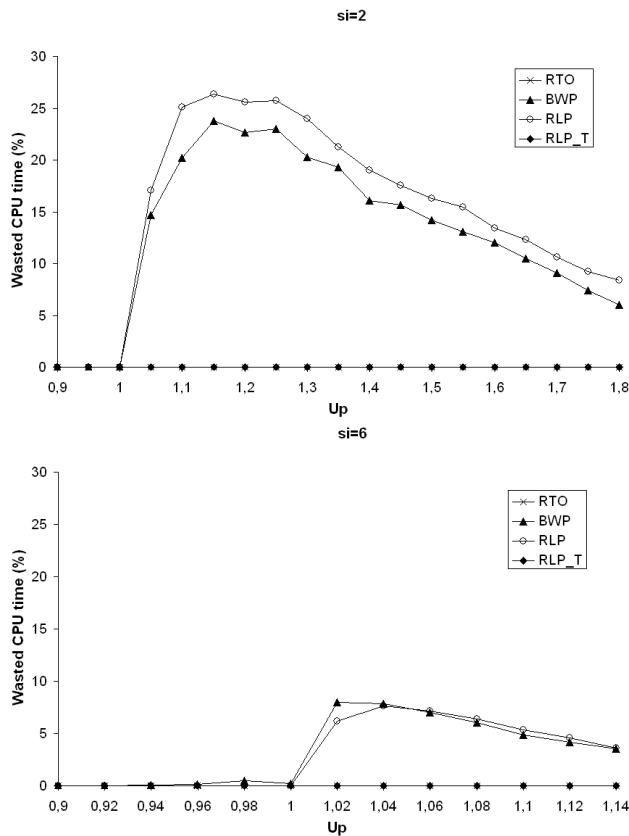
Finally, other tests not reported here show that, higher is the skip parameter more significant is the advantage of RLP/T over the other algorithms.

## 6.3. Experiment 2

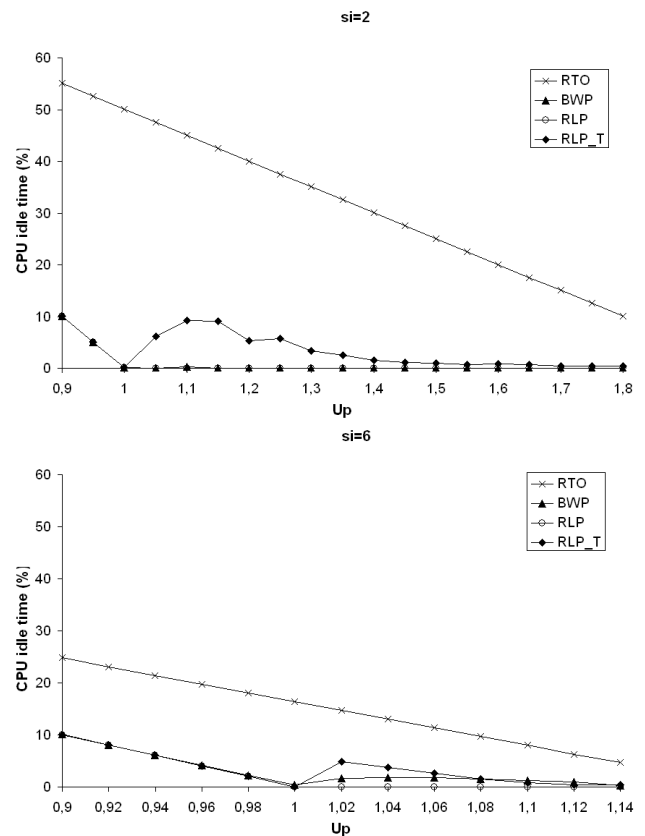
It seemed wise to perform measurements about the CPU time wasted in incomplete executions of blue tasks. Simulation results for  $s_i = 2$  and  $s_i = 6$  are depicted in Figure 7.

Wasted CPU time is equal to zero for RTO since the algorithm schedules only red tasks. For RLP/T, the wasted CPU time is also equal to zero, whatever is the periodic load. This has been achieved thanks to the admission test implementation which prevents the abortion of blue tasks. A blue task is accepted if and only if it will be able to complete before its deadline.

With respect to BWP and RLP, the wasted CPU time is always positive once the system is overloaded ( $U_p > 1$ ). For  $U_p = 115\%$  and  $s_i = 2$ , BWP et RLP involve the greatest amount of wasted CPU time, namely 24% et 26% respectively. Beyond that load, BWP and RLP curves present a decline. This is due to the fact that when the system is highly overloaded, it means that there are more red tasks to execute, hence less available CPU time for the execution of blue tasks. In addition, results not reported here show that the wasted CPU time is all the less significant as the skip parameters are higher.



**Figure 7.** Wasted CPU time for high ( $s_i = 2$ ) and low ( $s_i = 6$ ) skips



**Figure 8.** CPU idle time for high ( $s_i = 2$ ) and low ( $s_i = 6$ ) skips

## 6.4. Experiment 3

Finally, we made a performance comparison on the basis of another criterion : the CPU idle time ratio (*i.e.* the ratio of time during which the processor is not processing any task). This measure represents the system ability to face a dynamic surplus of processing (*e.g.* the arrival of an aperiodic task). Simulation results for  $s_i = 2$  et  $s_i = 6$  are presented in Figure 8.

First, let us remark that the CPU idle time ratio under RTO is the biggest one. This one declines in a linear fashion according to the periodic load  $U_p$  applied to the system. In the case  $s_i = 2$ , it varies from 55% for  $U_p = 90\%$ , to 10% for  $U_p = 180\%$ . Note the singular points of the curves  $s_i = 2$  et  $s_i = 6$ : when  $U_p = 100\%$ , idle time ratios are respectively equal to  $\frac{1}{2} = 50\%$  and  $\frac{1}{6} = 16.7\%$ , thus corresponding exactly to the allowed skip ratios.

As regards BWP, RLP and RLP/T, idle time ratios are identical when the system is underloaded ( $U_p < 100\%$ ). They become positive (idle time = 10% for  $U_p = 90\%$ ) and decline in a linear fashion until reaching a zero value for  $U_p = 100\%$ .

However, when the system is overloaded ( $U_p > 100\%$ ), results differ. RLP doesn't involve CPU idle time whatever are the skip parameters. We observe that BWP involves a low idle time ratio only for low skip ratios. As regards RLP/T, it seems the most performant keeping a non neglecting idle time ratio when the system is lightly overloaded. Indeed, the idle time ratio under RLP/T for  $s_i = 2$  and  $U_p = 115\%$  is equal to 9%. Moreover, even when the system is highly overloaded, RLP/T presents an idle time ratio which is low, certainly, but always positive.

As a matter of fact, all the experiments show that RLP/T seems the most suitable for facing transient overloads.

## 7. Implementation and validation issues

### 7.1. Integration in a Linux based operating system

There are two approaches to provide real-time performance in a Linux system: 1. Improving the Linux kernel preemption. 2. Adding a new software layer beneath Linux kernel with full control of interrupts and processor

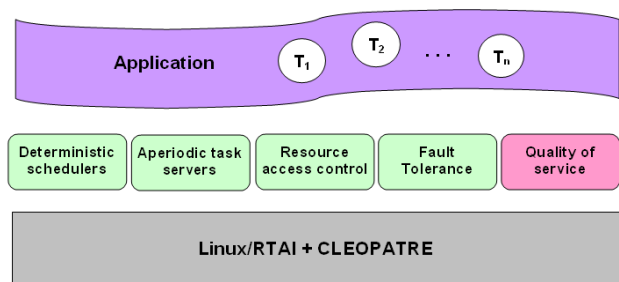
key features. This second approach is the one used by RTAI [6]. RTAI is the acronym of Real Time Application Interface. It was first started at the Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano by Professor Paolo Mantegazza. RTAI started as a variant of RTLinux in 1999. Recently, RTAI developers have replaced the underlying hardware control RTHAL, based on the RTLinux original code and patented by Victor Yodaiken [15], with a new technology called ADEOS “Adaptive Domain Environment for Operating Systems” [14]. The basic structure of RTAI is the same than RTLinux. A new software layer is beneath Linux kernel with full control of interrupts and processor key features. RTAI scheduler treats the Linux operating system kernel as the idle task. Linux only executes when there are no real time tasks to run, and the real time kernel is inactive. Linux processes can never block interrupts or prevent themselves from being preempted. In this way, it is possible to have a complete general purpose operating system running on top of a small predictable RTOS. RTAI provides an execution environment “below” the Linux kernel. One consequence of this is that real-time tasks can not use Linux services because deadlock or system inconsistencies may happen. To overcome this problem, a FIFO-based mechanism can be used to communicate threads in both layers.

An optional tracer module (Linux Trace Toolkit) can be included in the system. The tracer will register all the relevant system events and user-defined ones. LTT provides developers with all of the information necessary to reconstruct a system’s behavior over a certain period of time and find logical and temporal bugs of the system or the application. Linux and RTAI both lack some important facilities needed in real-time systems.

## 7.2. The CLEOPATRE framework

CLEOPATRE, that stands for Open Components on Shelves for Embedded Real-time Applications, is a french national project, based on Open Source, which aims to provide an integrated execution environment for embedded real-time applications with flexible and portable components. So, it was mandatory to design the components taking as starting point a given RTOS. Without a basic infrastructure, it was not possible to develop new functionalities. RTAI was 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.

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 9).



**Figure 9.** The CLEOPATRE framework

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 [1].

RTO, BWP, RLP and RLP/T 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.

## 7.3 Tests and validation criteria

To validate the correct implementation of this component two types of tests have been developed:

- Conformance tests. These tests have checked the correct behavior of the API.
- Overhead tests. These tests have validated the following criteria (for a Pentium III 400 MHz): overhead < 400 microseconds incurred with RLP/T and overhead < 100 microseconds incurred with RLP, for a set of 20 tasks whose execution was simulated during 34 seconds.

Details of implementation and tests can be found in [7].

## 8. Conclusions and Future Work

It is generally accepted that occasionally missing some firm deadlines is acceptable and is not considered as a fail-

ure. Hence, during transient overloads, some of the non-critical tasks can be dropped without jeopardizing system correctness. Most research in the area of scheduling firm tasks assume that there is no requirement on the minimum number of tasks that must successfully complete. In this paper, we have focused on scheduling firm periodic tasks which may skip occasionally. In other words, system correctness is maintained if, for all tasks, the scheduler enforces the skip factor while maximizing the robustness (*i.e.* the task completion ratio). We have suggested an overload scheduling algorithm, RLP, for a periodic workload that complies to the Skip-Over model and we have presented a variant of RLP, called RLP/T based on an admission control mechanism. Tasks having better chances to successfully complete with RLP/T, robustness of the system *i.e.* the completion ratio is increased. We have evaluated RLP/T by conducting a simulation-based performance analysis. The results show that the performance of firm tasks gracefully degrades as the load increases. We have presented CLEOPATRE, a library of free software components for the design of a high variety of embedded real-time systems to cover several classes of applications. Overload scheduling strategies including RLP and RLP/T are provided by one shelf of the library and are the key features needed to provide a powerful and usable RTOS, that permits to manage overload conditions for firm real-time systems. We intend to present, in future papers, the feasibility of using the RLP/T algorithm for scheduling skippable periodic tasks in presence of soft and hard aperiodic tasks under resource access constraints.

## References

- [1] <http://cleopatre.rts-software.org>.
- [2] R.-H. Campbell, K.-H. Horton, and G.-G. Belford. Simulations of a fault-tolerant deadline mechanism. *Digest of papers FTcs-9*, pages 95–101, 1979.
- [3] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, October 1989.
- [4] H. Chetto and M. Chetto. An adaptive scheduling algorithm for fault-tolerant real-time systems. *Software Engineering Journal*, May 1991.
- [5] G. Koren and D. Shasha. Skip-over algorithms and complexity for overloaded systems that allow skips. *16th IEEE Real-Time Systems Symposium*, 1995.
- [6] P. Mantegazza, E. Bianchi, L. Dozio, M. Angelo, and D. Beal. Diapm. rtai programming guide 1.0. *Lineo Inc.*, 2000.
- [7] A. Marchand. *Ordonnancement temps réel avec contraintes de qualité de service - De la théorie à l'intégration*. PhD thesis, University of Nantes (France), October 2006.
- [8] A. Marchand and M. Silly-Chetto. Qos scheduling components based on firm real-time requirements. *ACS/IEEE International Conference on Computer Systems and Applications*, January 2005.
- [9] A. Marchand and M. Silly-Chetto. Rlp: Enhanced qos support for real-time applications. *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, August 2005.
- [10] A. Marchand and M. Silly-Chetto. Dynamic real-time scheduling of firm periodic tasks with hard and soft aperiodic tasks. *The Journal of Real-Time Systems*, 32(1-2):21–47, February 2006.
- [11] M. Silly. The edl server for scheduling periodic and soft aperiodic tasks with resource constraints. *The Journal of Real-Time Systems*, Kluwer Academic Publishers, 17:1–25, 1999.
- [12] M. Silly, H. Chetto, and N. Elyounsi. An optimal algorithm for guaranteeing sporadic tasks in hard real-time systems. *IEEE Symposium on Parallel and Distributed Processing*, pages 578–585, 1990.
- [13] R. West and C. Poellabauer. Analysis of a window-constrained scheduler for real-time and best-effort packet streams. *21st IEEE Real-Time Systems Symposium*, November 2000.
- [14] K. Yaghmour. Adaptive domain environment for operating systems. *Opsys Inc.*, 2001.
- [15] V. Yodaiken. The rtlinux approach to real-time. *FSMLabs Inc.*, 2004.