On QoS Scheduling Services for Real-Time Periodic Tasks

A. Marchand, M. Silly-Chetto

Institut de Recherche en Communications et Cybernétique de Nantes
UMR CNRS 6597 - 1, rue de la Noë - BP 92 101
44321 NANTES Cedex 03, France

Abstract

In this paper, we deal with dynamic scheduling components integrating new Quality of Service (QoS) functionalities into a Linux-based real-time operating system. In our approach, periodic tasks allow occasional deadline violations. The minimal QoS level tolerated by each periodic task, i.e. the ratio of periodic task instances which complete before their deadline, is set by the application programmer. The work stated here provides two on-line scheduling algorithms, namely RLP and RLP/T in order to enhance the QoS observed for periodic tasks. These novel scheduling techniques rely on both the Skip-Over task model and the EDL (Earliest Deadline as Late as possible) scheduling strategy. Simulation results show the performance of RLP and RLP/T with respect to basic Skip-Over algorithms. We also present the integration of these QoS scheduling services into CLEOPATRE open-source component library, a patch to Linux/RTAI. Finally, we evaluate the performance of these new QoS components quantifying the scheduling overheads observed under Linux/RTAI.

Key words:
real-time, dynamic scheduling, quality of service, earliest deadline, periodic tasks, Linux-based systems

1 Introduction

Software environments, and more precisely operating systems have still difficulties to meet the special demands of multimedia applications. In particular, multimedia applications have real-time constraints which are not handled properly by general-purpose operating systems. In order to meet the requirements imposed by multimedia applications on processor scheduling, we have to turn to the temporal stringency of real-time systems. Real-time systems are those in which the time at which the results are produced is important. The correctness of the result of a task is not only related to its logic correctness, but also to when the results occur.

Traditional classification of real-time systems stands for three classes to characterize the real-time requirement of such systems : hard, soft and firm. 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 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 firm systems, tasks are also 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 firm 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 (Hamdaoui and Ramanathan, 1995) introduced the concept 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 (Koren and Shasha, 1995) 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 contraints. The scope of the paper is to maximize the QoS of periodic tasks by maximizing the number of instances which complete before their deadline. The remainder of this paper is organized in the following manner. Section 2 presents relevant background material about both the Skip-Over model and the EDL scheduling algorithm. More particularly, we give the definition of RTO and BWP scheduling algorithms, which are based on the Skip-Over model. The functioning and optimality of the EDL algorithm is also outlined. Further, we describe the proposed algorithms, namely RLP and RLP/T, as an enhancement of the BWP algorithm, based on the EDL scheduling mechanism. The performance analysis of both RLP and RLP/T, in terms of task completions, is reported in secton 5. In sections 6 and 7 respectively, we describe the integration of these QoS components into Linux/RTAI and evaluate them. Finally, in section 8, we summarize our contribution.

2 Theoretical background

2.1 The Skip-Over model

We are here interested in 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 and that 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, which gives the tolerance of this task to missing deadlines. 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 equivalent to a hard periodic task. One can view the skip parameter as a QoS metric (the higher s_i, the better the quality of service).

A task T_i is divided into instances where each instance occurs during a single period of the task. Every instance of a task can be red or blue (Koren and Shasha, 1995). 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.

2.1.1 Red Tasks Only (RTO) algorithm

The first algorithm 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, while blue ones are always rejected. Deadline ties are broken in favor of the task with the earliest release time. In the deeply red model where all tasks are synchronously activated and the first $s_i - 1$ instances of every task T_i are red, this algorithm is optimal. RTO is illustrated in Figure 1 using the task set $\mathcal{T} = \{T_0, T_1, T_2, T_3, T_4\}$ of five periodic tasks whose parameters are described in Table 1. Tasks have uniform skip parameter $s_i = 2$ and the total processor utilization factor $U_p = \sum \frac{c_i}{p_i}$ is equal to 1.15.

Task	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
A basic periodic task set

As we can see, the distance between every two skips is exactly s_i periods, thus offering only the minimal guaranteed QoS level for periodic tasks.

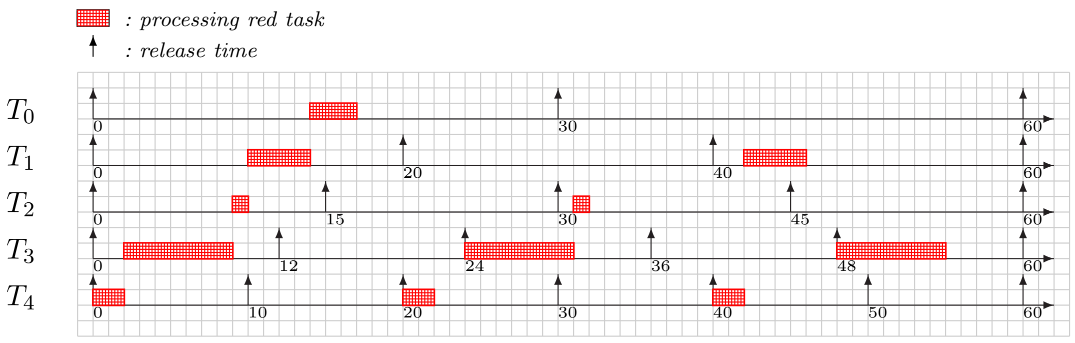

Fig. 1. RTO scheduling algorithm ($s_i = 2$)

2.1.2 *Blue When Possible (BWP) algorithm*

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 that sense, it operates in a more flexible way. Deadline ties are still broken in favor of the task with the earliest release time. Figure 2 shows an illustrative example of BWP scheduling using the task set previously described in Table 1.

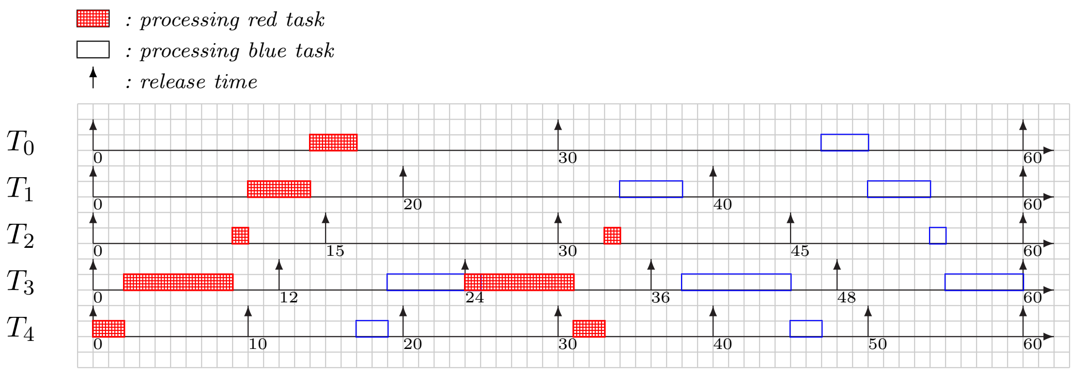

Fig. 2. BWP scheduling algorithm ($s_i = 2$)

Compared with RTO, more task instances complete successfully with BWP. We observe that five violations of deadline relative to blue task instances occur at time instants $t = 24$ (task T_3), $t = 30$ (tasks T_2 and T_4) and $t = 60$ (tasks T_3 and T_4), thus reducing the QoS.

2.2 *The EDL algorithm*

The definition of the Earliest Deadline as Late as possible (EDL) algorithm makes use of some results presented by Chetto and Chetto in (Chetto and Chetto, 1989). Under EDL, periodic tasks are scheduled as late as possible.

An accurate characterization of the idle times during which the processor is not occupied is necessary. The authors introduced an *availability function* f_Y^X defined with respect to a task set Y and a scheduling algorithm X.

$$\begin{aligned} f_Y^X(t) &= 1 \quad \textit{if the processor is idle at } t \\ &= 0 \quad \textit{otherwise} \end{aligned} \tag{1}$$

So, for any instants t_1 and t_2, value of $\int_{t_1}^{t_2} f_Y^X(t)dt$ denoted by $\Omega_Y^X(t_1, t_2)$ gives the total idle time in $[t_1, t_2]$. f_Y^{EDL} can be described by means of the following two vectors:

- $\mathcal{K}$, called *static deadline vector*, represents the times at which idle times occur and is constructed from the distinct deadlines of periodic tasks.
- $\mathcal{D}$, called *static idle time vector*, represents the lengths of the idle times relating to time instants of vector $\mathcal{K}$.

The complexity of the EDL algorithm is $O(Kn)$ where n is the number of periodic tasks, and K is equal to $\lfloor \frac{R}{p} \rfloor$, where R is the longest deadline, and p is the shortest period (Silly, 1999). We also recall the fundamental property relative to the optimality of EDL (Chetto and Chetto, 1989):

Theorem 1 *Let X be any preemptive scheduling algorithm and $\mathcal{A}$ a set of independent aperiodic tasks. For any instant t,*

$$\Omega_{\mathcal{A}}^X(0,t) \leq \Omega_{\mathcal{A}}^{EDL}(0,t) \tag{2}$$

We give now an illustrative example of the computation of the idle times performed by EDL. Consider the periodic task set $\mathcal{T} = \{T_1, T_2\}$ consisting of two periodic tasks $T_1(3, 10)$ and $T_2(3, 6)$. The $f_{\mathcal{T}}^{EDL}$ computation produced at time zero is described in Figure 3.

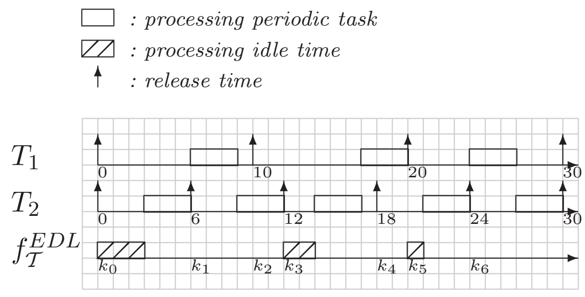

Fig. 3. $f_{\mathcal{T}}^{EDL}$ computation produced at time zero

The authors in (Chetto and Chetto, 1989) described how the EDL algorithm can be applied, first to the decision problem that arises when a sporadic time critical task occurs and requires to be run at an unpredictable time and secondly, to the scheduling problem that arises in a fault tolerant system using the Deadline Mechanism (Liestman and Campbell, 1986) for which each task implements primary and backup copies (the processor time reserved for the execution of the backup copies is realized with EDL and is reclaimed as soon as the primary task executes successfully).

In next sections, we are interested in using EDL first to simulate a schedule (RLP implementation) and then to derive a measure required for deciding whether a blue task can be accepted (RLP/T implementation).

3 Red tasks as Late as Possible (RLP)

3.1 Algorithm outline

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 behaviour:

(1) 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.
(2) if blue task instances are present in the system, these ones 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 by a variant of the EDL algorithm taking skips into account (Marchand and Silly-Chetto, 2006). In the EDL schedule established at time τ, we assume that the instance following immediately a blue instance which is part of the current periodic instance set at time τ, is red. Indeed, none of the blue task instances is guaranteed to complete within its deadline. Moreover, Silly in (Silly, 1999) 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).

3.2 Illustrative example

Consider once again the periodic task set $\mathcal{T}$ defined in Table 1. The relating RLP scheduling is illustrated in Figure 4.

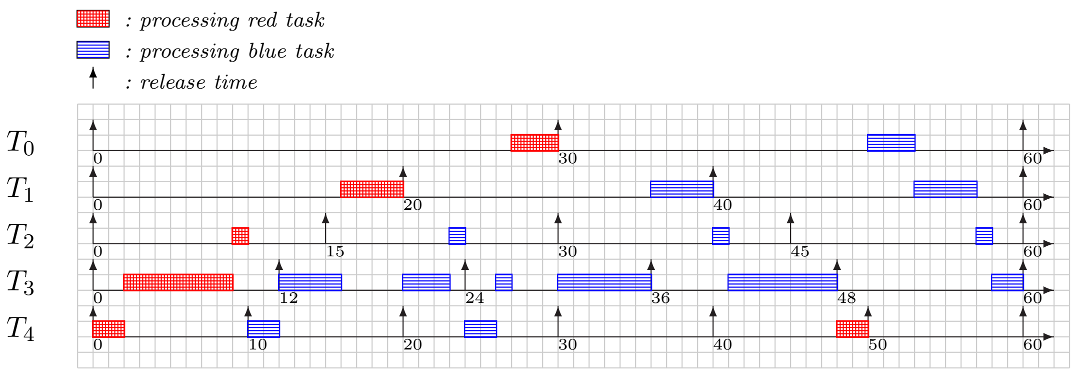

Fig. 4. RLP scheduling algorithm ($s_i = 2$)

In this example, we can see that, thanks to RLP scheduling, the number of violations of deadline relative to blue task instances has been reduced to three. 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 failed to complete within its deadline in the BWP case (see Figure 2), has enough time to succeed in the RLP case, 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 hyperperiod (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.

4 Red tasks as Late as Possible with blue acceptance test (RLP/T)

4.1 Algorithm outline

Red tasks as Late as Possible with blue acceptance test (RLP/T) algorithm is designed to maximize the QoS of periodic task sets defined under skip constraints.

It acts as follows: red tasks enter straight the system at their arrival time whereas blue tasks integrate the system upon acceptance. Once they have been accepted, blue tasks are scheduled as soon as possible together with red tasks. Upon acceptance, blue tasks are again of same importance beside red tasks. Deadline ties are always broken in favor of the task with the earliest release time.

Whenever a new blue task enters the system, the idle times are computed using the EDL scheduler. In the EDL schedule established at time τ, we assume that the instance following immediately a blue instance which is part of the current periodic instance set at time τ, is also blue. Indeed, all blue task instances previously accepted at τ are guaranteed by the schedulability test they passed successfully. This one checks whether there are enough idle times to accommodate the new blue task within its deadline, as described in the following section.

4.2 Acceptance test of blue tasks under RLP/T

Now, we are ready to present the new feasibility test algorithm for the RLP scheduling scheme which, given any occurring blue task B is capable of answering the question "Can B be accepted ?". Notice that B will be accepted if and only if there exists a valid schedule, *i.e.*, a schedule in which B will execute by its deadline while red periodic tasks and blue tasks previously accepted, will still meet their deadlines. Let τ be the current time which coincides with the arrival of a blue task B. Upon arrival, task $B(r, c, d)$ is characterized by its release time r, its execution time c, and its deadline d, with $r + c \leq d$. We assume that the system supports several blue tasks at time τ. Each of them has been accepted before τ and has not completed its execution at time τ. Let denote by $\mathcal{B}(\tau) = \{B_i(c_i(\tau), d_i), i{=}1 \text{ to } blue(\tau)\}$ the blue task set supported by the machine at τ. Value $c_i(\tau)$ is called dynamic execution time and represents the remaining execution time of B_i at τ. A deadline occurs at d_i. We assume that $\mathcal{B}(\tau)$ is ordered such that $i < j$ implies $d_i \leq d_j$.

The acceptance test of blue tasks within a system involving RLP skippable tasks presented below in Theorem 2, is based on the one established by Silly and al. (Silly et al., 1990) for the acceptance of sporadic requests occurring in a system consisting of basic periodic tasks (*i.e.*, without skips).

Theorem 2 *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:*

$$\delta_i(\tau) = \Omega^{EDL}_{\mathcal{T}(\tau)}(\tau, d_i) - \sum_{j=1}^{i} c_j(\tau) \tag{3}$$

$\delta_i(\tau)$ is called slack of task B_i at time τ which represents the maximum units of time during which the task could not be served by the processor without missing its deadline. $\Omega^{EDL}_{\mathcal{T}(\tau)}(\tau, d_i)$ 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 $\sum_{j=1}^{i} c_j(\tau)$

The procedure that implements the acceptance test calls for the EDL algorithm for the computation of the total idle times which will be used to compute the slack of blue tasks. Then, this slack is compared to zero. Thus, the acceptance test proposed in this paper runs in $O(\lfloor \frac{R}{p} \rfloor n + blue(\tau))$ in the worst-case, where n is the number of periodic tasks, R is the longest deadline, p is the shortest period, and $blue(\tau)$ denotes the number of active blue tasks at time τ, whose deadline is greater or equal to the deadline of the occurring task. Note that this acceptance test could be implemented in $O(n + blue(\tau))$ by considering and maintaining to update additional data structures using slack tables, as proved in (Tia et al., 1994).

4.3 Illustrative example

RLP/T scheduling is illustrated in Figure 5 with the periodic task set $\mathcal{T}$ defined in Table 1.

It is easy to see that RLP/T improves on both RLP and BWP. Only two violations of deadline relative to blue task instances are observed: at time instants $t = 40$ (task T_4) and $t = 60$ (task T_3). 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_3 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

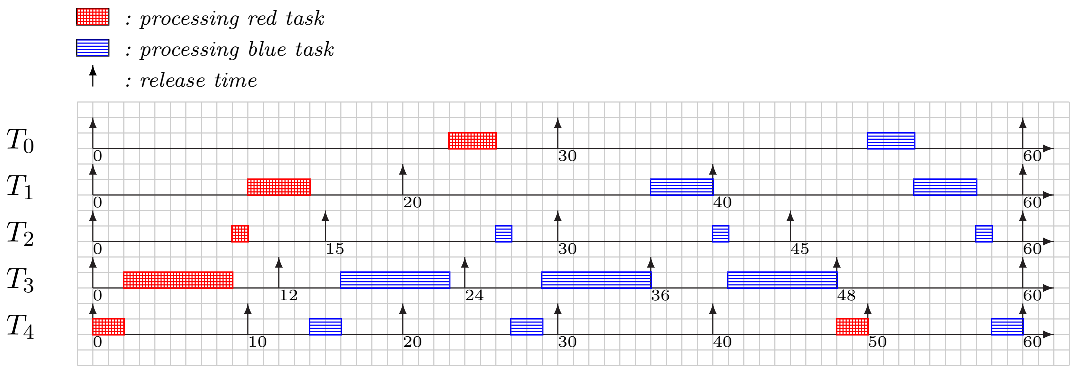

Fig. 5. RLP/T scheduling algorithm ($s_i = 2$)

the successful completion of T_4 blue instance released at time $t = 50$.

In next section we quantify more precisely the gain of performance of RLP/T upon RLP, BWP and RTO.

5 Performance evaluation using a simulation model

In this section, we summarize the results of simulation studies which compare the performance of the different QoS scheduling algorithms. The objective is to maximize the QoS level of periodic tasks, i.e., the ratio of periodic tasks which complete before their deadline.

Experiments also evaluate the impact of the skip value for each algorithm, namely RTO, BWP, RLP and RLP/T.

5.1 Simulation context

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 contraints with uniform s_i. Their worst-case execution time depends 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.

5.2 Varying the periodic load

Measurements rely on the ratio of periodic tasks which complete before their deadline. The evaluation is done varying the periodic load U_p. The results obtained for $s_i = 2$ (one instance every two can be aborted) and $s_i = 6$ (one instance every six can be aborted) are described on Figure 6 and Figure 7 respectively.

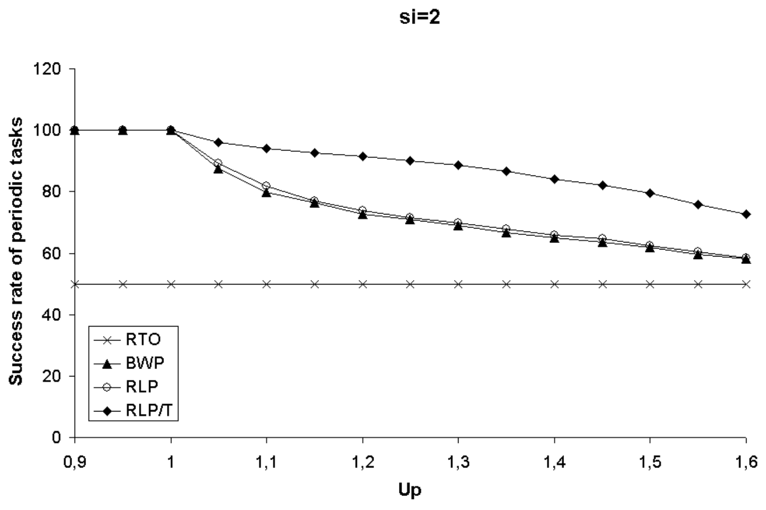

Fig. 6. QoS of periodic tasks with low skip parameters ($s_i = 2$)

From the graphs, we can say that BWP, RLP and RLP/T outperform the RTO model in which the QoS level is still constant whatever is the periodic load applied. For $s_i = 6$, the QoS remains constant at a rate of 5/6=83%. The advantage of RLP over BWP is slight for low skip parameters, and more significant for high skip parameters. We note that the performance of BWP and RLP is dramatically worse than the one achieved by RLP/T. This result was expected because both BWP and RLP attempt to schedule blue instances that have not enough time for completing within their deadlines. This wasted time is not saved for executing other blue instances with closer deadline. In contrast, RLP/T finds a way of saving this CPU time by implementing an acceptance test for blue instances. We can observe that this gain of performance is all the more significant as the periodic load U_p is higher. For instance, in Figure 6, for $U_p \geq 120\%$, RLP/T enjoys more than factor $\frac{1}{4}$ success rate advantage over BWP. Moreover, we observe a very low gradient for the RLP/T curve which is not the case for other models. For $U_p = 150\%$ and $s_i = 2$, QoS levels for RTO and RLP/T are respectively equal to 50% and 84%, which figures the great predominance of RLP/T over RTO.

The variation of the skip parameter value shows that, for wide loads, the QoS of periodic tasks is all the more improved with RLP/T that the QoS constraint is smaller. For instance, for $U_p = 110\%$, RLP/T applied to periodic tasks with $s_i = 2$ (see Figure 6) will successfully process twice as many periodic instances over BWP, as with periodic tasks with $s_i = 6$ (see Figure 7). As we can see, the major difference in the performance between RLP/T and BWP appears not only for heavy loads but also for small value of s_i.

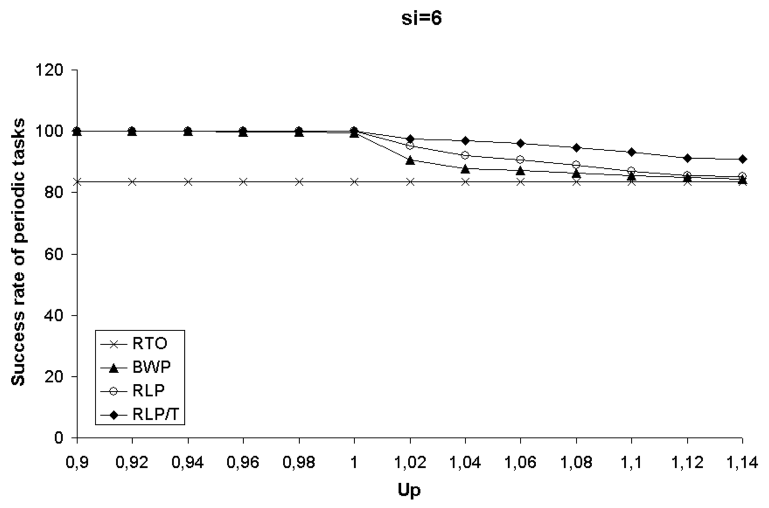

Fig. 7. QoS of periodic tasks with high skip parameters ($s_i = 6$)

6 Integration into a Linux-based system

6.1 CLEOPATRE library

The emergence of more and more complex industrial applications pointed out the need of academic research transfer into Linux based operating systems. In this context, 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) [1] in order to provide more efficient and better service to real-time applications. The purpose was to enrich the real-time facilities of real-time Linux versions, such as RTLinux (V. Yodaiken, 2004) or RTAI (Mantegazza et al., 2000). RTAI was the solution adopted for this project because we wanted the CLEOPATRE components to be distributed under the LGPL [2] license which is also the one used in the RTAI project.

The CLEOPATRE library whose framework is shown in Figure 8 offers selectable COTS (Commercial-Off-The-Shelf) components dedicated to dynamic scheduling, aperiodic task service, resource control access and fault-tolerance.

Scheduling components are based on two scheduling algorithms: Deadline Monotonic (DM) and Earliest Deadline First (EDF). Concerning the shelf that provides aperiodic tasks servicing, three servers have been implemented, in order to cope with soft and hard aperiodic tasks arrivals: Background server, Total Bandwidth server (TBS) (Caccamo et al., 1999), and Earliest Deadline

[1] work supported by the French research office, grant number 01 K 0742
[2] Lesser General Public License

as Late as possible server (EDL). Five resource management protocols are available: FIFO (First In First Out), Priority, SPP (Super Priority Protocol), PIP (Priority Inheritance Protocol) and PCP (Priority Ceiling Protocol) (Sha et al., 1990). A fault-tolerance mechanism has been implemented: the Deadline Mechanism (Liestman and Campbell, 1986). The problem of the overload management has also been covered by the implementation of the Imprecise Computation model (Chung et al., 1990).

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.

It is responsible for managing low-level TaskType (extended RTAI RT_TASK task descriptor) tasks through various functions (TCL_CREATE, TCL_DESTROY, TCL_KILL, TCL_READY, TCL_BLOCK and TCL_SCHEDULE) described in (Garcia et al., 2003).

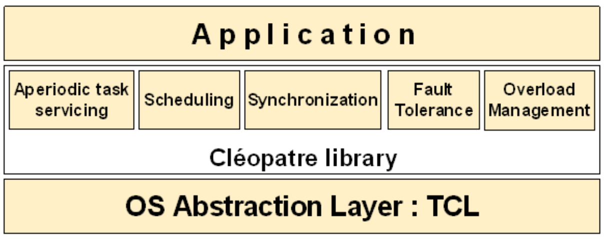

Fig. 8. The Cleopatre framework

Dynamic scheduling of periodic tasks with skips, namely RTO, BWP and RLP/T algorithms, have been put into an additional shelf called *Quality Of Service*.

6.2 Data Structure

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 the task function*/
  TaskType TCL_task;            /*low-level task descriptor*/
  TimeType critical_delay;      /*critical delay*/
  TimeType period;              /*period of activation*/
  TimeType release_time;        /*time at which the task is released*/
  unsigned int max_skipvalue;     /*maximum tolerance to skips*/
  unsigned int current_skipvalue;/*dynamic skip parameter*/
  unsigned int current_shift;     /*shift compared with an RTO sequence*/
  unsigned int slack;             /*slack of the task*/
};
```

The user interface for the QoS schedulers is given in Table 2.

QoS functions	Description
QoS_create	create a new real-time task
QoS_resume	resume a real-time task
QoS_wait	wait till next period
QoS_delete	delete a real-time task

Table 2
QoS schedulers' interface

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.

6.3 Algorithms

The scheduling of QoS tasks is performed in the QoS_schedule() function. The scheduling occurs on timer handler activation (each 8254 interrupt). In our implementation, the scheduler maintains three linked lists 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 put into the red_ready_list list are always performed before those present into the blue_ready_list list. At QoS_schedule() time, the currently running task is the default candidate to run next. A task is considered schedulable if it is not already running and it is enabled for dispatch on the CPU.

6.3.1 RTO algorithm implementation

In RTO module, the QoS_schedule() routine attempts to release tasks from the waiting_list list. If a task is found with a release_time parameter lesser than or equal to current time, then it is put into the red_ready_list list. The scheduling decision is in the worst-case in $O(N^2)$, where all the N tasks have to be released at the same time. The algorithmic description of RTO is given below:

```
**Algorithm 1**
QoS_schedule(t : current time)
begin
   /*Checking waiting_list in order to release tasks*/
   while (task=next(waiting_list)=not(∅))
         if (task→release_time<t)
            break
         endif
         Pull task from waiting_list
         Place task into red_ready_list
   endwhile
end
};
```

6.3.2 BWP algorithm implementation

The QoS_schedule() routine of BWP operates in two distinct phases. In the first phase, it examines blue_ready_list in order to abort, if necessary, blue tasks whose deadlines are greater than or equal to current time. The waiting_list list is scanned in the second phase so as to resume tasks whose release time is lesser than or equal to current time.

```
Algorithm 2
QoS_schedule(t : current time)
begin
   /*Checking blue_ready_list in order to abort tasks*/
   while (task=next(blue_ready_list)=not(∅))
         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
         Place task into waiting_list
   endwhile
   /*Checking waiting_list in order to release tasks*/
   while (task=next(waiting_list)=not(∅))
         if (task→release_time<t)
            break
```

```
          endif
          Pull task from waiting_list
          if (task→current_skipvalue<task→max_skipvalue)
             Place task into red_ready_list
          else
             Place task into blue_ready_list
          endif
   endwhile
end
};
```

6.3.3 RLP algorithm implementation

The QoS_schedule() routine of RLP proceeds also in three stages. In the first stage, it examines blue_ready_list in order to or abort one or several blue tasks which have reached their deadline. The waiting_list list is scanned in the second stage so as to resume tasks whose release time is lesser than or equal to current time. Red tasks are put in the red_ready_list list when there is no idle time at current time, contrary to blue ones released only when there is an idle time. In the last stage, it examines red_ready_list in order to suspend ready red tasks (released before current time), on the sole condition that there are tasks into the blue_ready_list list and there is an idle time at current time.

Algorithm 3

```
QoS_schedule(t : current time)
begin
   /*Checking blue_ready_list in order to abort tasks*/
   while (task=next(blue_ready_list)=not(∅))
          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
          Place task into waiting_list
   endwhile
   /*Checking waiting_list in order to release tasks*/
   while (task=next(waiting_list)=not(∅))
          if (task→release_time<t)
             break
          endif
          if ((task→current_skipvalue<task→max_skipvalue) and (f_EDL(t)=0))
             Pull task from waiting_list
```

```
            Place task into red_ready_list
        else
            if (blue_ready_list=∅)
                Compute f_EDL
            endif
            if (f_EDL(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_ready_list)=not(∅))
        if (blue_ready_list=not(∅)) and (f_EDL(t)!=0)
            Pull task from red_ready_list
            Place task into waiting_list
        endif
    endwhile
end
};
```

6.3.4 RLP/T algorithm implementation

The waiting_list list is scanned so as to resume tasks whose release time is lesser than or equal to current time. Red tasks are directly put in the red_ready_list list, whereas blue tasks are accepted according to the result of the acceptance test which depends on the computation of the EDL schedule. Note that upon acceptance, since blue tasks are scheduled as soon as possible together with red tasks, they are considered as red tasks and put in the same ready list.

Algorithm 4

```
QoS_schedule(t : current time)
begin
    /*Checking waiting_list in order to release tasks*/
    while (task=next(waiting_list)=not(∅))
        if (task→release_time<t)
            break
        endif
        if (task→current_skipvalue<task→max_skipvalue)
            Pull task from waiting_list
            Place task into red_ready_list
        else
            Compute f_EDL
        if (Blue_acceptance_test(task, f_EDL))
          Pull task from waiting_list
          Place task into red_ready_list
        endif
      endif
  endwhile
end
};
```

7 Performance evaluation of the QoS components

Some few points should be considered when trying to use a scheduler for real-time or embedded applications. For embedded applications, the memory footprint and disk footprint are generally key issues. For real-time applications, there is the requirement to meet task deadlines in addition to the logical correctness of the results. This implies an evaluation of the execution speed of the scheduler, so as to check the non-interference of the scheduling algorithm with respect to task completions.

7.1 Footprints

Memory and disk footprints of the QoS scheduling components are summarized in Table 3.

QoS scheduling component	Hard disk size (KB)	Memory size (KB)
RTO	3.2	2.3
BWP	4.1	3.2
RLP	9.7	7.6
RLP/T	13.3	10.8
dependent on core modules	Hard disk size (KB)	Memory size (KB)
RTAI	27.2	23
TCL	34.8	27.1

Table 3
Footprints of QoS components

The minimal footprint for an application involving QoS periodic tasks is 52.4 KB in memory (65.2 KB in hard disk), including RTAI, TCL interface and RTO scheduler. The maximum footprint is reached if RLP/T scheduler is loaded : 60.9 KB in memory (75.3 KB in hard disk). In both cases, we can see that this application can be easily supported on an embedded device holding a floppy disk or a compact flash memory.

We have made some experiments with the implementation to make a quantitative evaluation of the overhead introduced by the QoS schedulers. Theses tests consist of measuring the overhead introduced when scheduling different number of tasks (5, 10, 15, 20,...) with periods of 10 milliseconds each one. Periods of all tasks are harmonic, leading up to a hyperperiod of 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.

The overhead we show for the QoS scheduling components (see Figure 9), indicates the amount of time spent performing scheduling tasks. As it can be seen from Figure 9, the overhead of the QoS schedulers scales with the number of installed tasks.

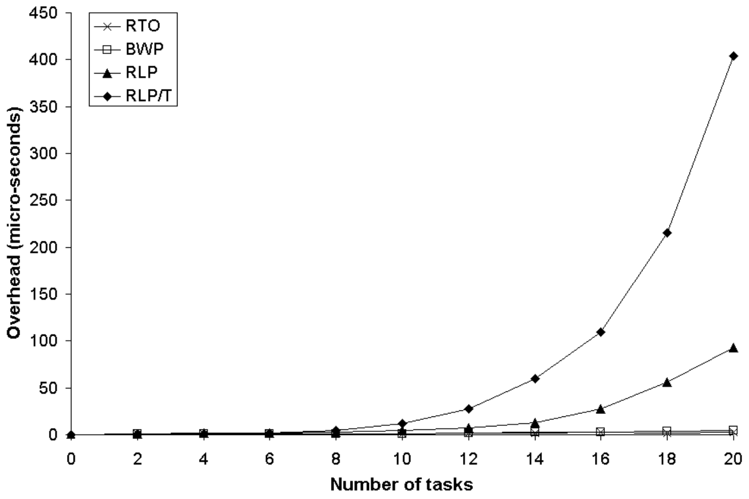

Fig. 9. Dynamic overhead of RTO, BWP, RLP and RLP/T schedulers

RTO runtime overhead is directly linked to the low complexity but also to the poor performance of its scheduling algorithm. We note that BWP scheduling runtime overhead is quite as low as RTO runtime overhead. The curve obtained for RLP scheduling algorithm is mainly due to the amount of time spent on the EDL sequence establishment (performed only when a blue task is released while there was no previous blue task in the system). RLP/T scheduling runtime overhead dramatically increases with the number of tasks. This is due to the idle time computation performed by RLP/T each time a blue task instance is released or completed. For 6 periodic tasks, it reaches the runtime overhead obtained for 20 periodic tasks with RTO scheduler. In practice, this efficient algorithm in terms of the maximization of the QoS offered to periodic tasks, will be interesting to use in applications involving less than 15 tasks.

7.3 Checking results with Linux Trace Toolkit (LTT)

Linux Trace Toolkit (LTT) is an effective tool for recording critical system information which provides users with a unique view of the system's behavior with minimal performance overhead (Ramya et al., 2004). More particularly, LTT presents the task execution in a graphical way. In our case, we will be able to display the task scheduling performed by the different QoS components integrated into Linux/RTAI.

Consider the periodic task set $\mathcal{T}$ defined in Table 1. The program implemented under Linux/RTAI for the testing of the QoS components is described below:

```
/*———Headers of all required components———*/
#include <TCL.h>
#include <QoS.h>
#include <simul.h>
/*——————Timer clock period (10ms)———-*/
#define TIMERTICKS 10000000
/*—————-Declaration of QoS tasks—————-*/
QoSTaskType T0;
QoSTaskType T1;
QoSTaskType T2;
QoSTaskType T3;
QoSTaskType T4;
/*—————Code description of QoS tasks———*/
void CodeT0() {simul.wait(3);}
void CodeT1() {simul.wait(4);}
void CodeT2() {simul.wait(1);}
void CodeT3() {simul.wait(7);}
void CodeT4() {simul.wait(2);}
/*—————Application initialization—————*/
int init_module(void)
{
  TCLCreateType create={0, 2000, 0, 0};
  /*************initializing QoS tasks*************/
  QoS.create(&T0, CodeT0, 3, 30, 30, 2, create);
  QoS.create(&T1, CodeT1, 4, 20, 20, 2, create);
  QoS.create(&T2, CodeT2, 1, 15, 15, 2, create);
  QoS.create(&T3, CodeT3, 7, 12, 12, 2, create);
  QoS.create(&T4, CodeT4, 2, 10, 10, 2, create);
  /**************resuming QoS tasks**************/
  QoS.resume(&T4,100);
  QoS.resume(&T3,100);
```

```
   QoS.resume(&T2,100);
   QoS.resume(&T1,100);
   QoS.resume(&T0,100);
   /***********switching to real-time mode**********/
   TCL.begin(TIMERTICKS, 20000);
   return 0;
}
/*——————Application ending——————-*/
void cleanup_module(void)
{
   /***************deleting QoS tasks***************/
   TCL.end();
}
```

Note that this program is equally used for RTO, BWP, RLP and RLP/T. The distinction is made at the dynamic loading stage when the user selects the QoS scheduling module he wants to use for its application.

Results obtained under LTT for RTO, BWP, RLP and RLP/T scheduling components are presented in Figure 10, 11, 13 and 13 respectively. On the left, the list of all tasks that existed during the trace is presented. On the right, horizontal lines indicate the time spent executing code belongs to the entity in the task list at the same height. A vertical transition means the flow of control has been passed to another entity.

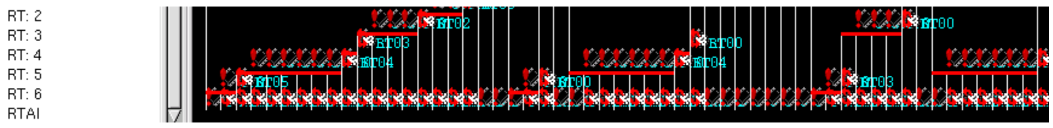

Fig. 10. RTO scheduling displayed with LTT

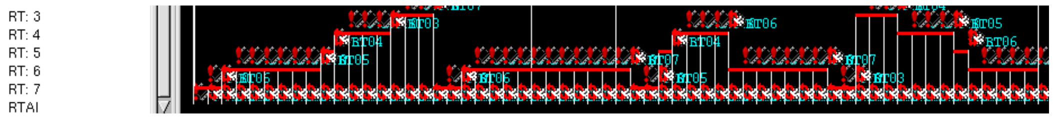

Fig. 11. BWP scheduling displayed with LTT

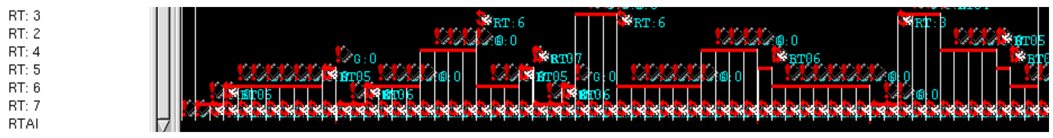

Fig. 12. RLP scheduling displayed with LTT

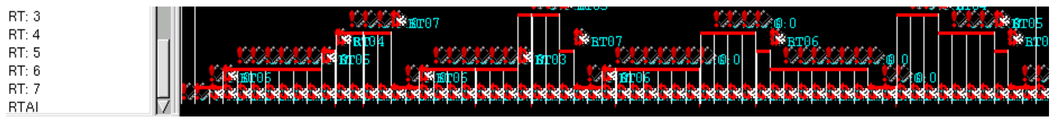

Fig. 13. RLP/T scheduling displayed with LTT

From the results, we can observe than actual scheduling performed under Linux/RTAI for RTO, BWP, RLP and RLP/T scheduling components conforms to their theoretical behavior previously shown in the paper in Figure 1, 2, 4 and 5.

8 Conclusions

This paper pointed out the need of more flexible scheduling solutions for real-time applications dealing with multimedia and active monitoring systems. Our main contribution was actually to provide scheduling algorithms, namely RLP and RLP/T, which enhance the QoS of periodic tasks that allow skips. Simulation results show that the improvements with these new scheduling algorithms are quite significant. Finally, these new QoS functionalities are available under Linux/RTAI as CLEOPATRE[3] components which can be dynamically loaded. Our future work includes extending these QoS scheduling algorithms to multiprocessor systems.

References

Marchand, A., Silly-Chetto, M., to appear in 2006. Dynamic Real-Time Scheduling of Firm Periodic Tasks with Hard and Soft Aperiodic Tasks. Journal of Real-Time Systems.

Chetto, H., Chetto, M., 1989. Some Results of the Earliest Deadline Scheduling Algorithm. In: Proceedings of the IEEE Transactions on Software Engineering, Vol. 15, No. 10, pp 1261-1269.

Caccamo, M., Lipari, G., Buttazzo, G., 1999. Sharing resources among periodic and aperiodic tasks with dynamic deadlines. In: Proceedings of the 20th IEEE Real-Time Systems Symposium.

Chung, J-Y., Liu, J-W-S., Lin, K., 1990. Scheduling periodic jobs that allow imprecise results. In: Proceedings of the IEEE Transactions on Computers, Vol.39, No.9, pp 1156–1174.

Garcia, T., Marchand, A., Silly-Chetto, M., 2003. Cleopatre: a R&D project for providing new real-time functionalities to Linux/RTAI. In: Proceedings of the Fifth Real-Time Linux Workshop.

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

Koren, G., Shasha, D., 1995. Skip-Over Algorithms and Complexity for Over-

[3] http://www.cleopatre-project.org

loaded Systems that Allow Skips. In: Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS'95), Pisa, Italy.

Liestman, A-L., Campbell, R-H., 1986. A fault tolerant scheduling problem. In: Proceedings of the IEEE Transaction on Software Engineering, Vol.12, No.10, pp 1089–1095

Mantegazza, P. , Bianchi, E. , Dozio, L., Angelo, M., Beal, D., 2000. DIAPM. RTAI Programming Guide 1.0, Lineo Inc.

Ramya, B-B., Pavithra, V. , Thangaraju, B., 2004. Process Tracing with the Linux Trace Toolkit. Sys Admin magazine.

Silly, M., 1999. 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.

Silly, M., Chetto, H., Elyounsi, N., 1990. An Optimal Algorithm for Guranteeing Sporadic Tasks in Hard Real-Time Systems. In: Proceedings of the IEEE Symposium on Parallel and Distributed Processing, pp 578-585.

Sha, L., Rajkumar, R., Lehoczky, J-P., 1990. Priority inheritance protocols: An approach to real-time synchronization. IEEE Transactions on Computers, pp 1175-1185.

Tia, T., Liu, J., Sun, J., Ha, R., 1994. A Linear-Time Optimal Acceptance Test for Scheduling of Hard Real-Time Tasks, Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign,Urbana-Champaign, IL.

Yodaiken, V., 2004. The RTLinux Approach to Real-Time, FSMLabs Inc.