Proc. of the 15th IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA'09), Beijing, China, August 24-26, 2009.

Bounding the Maximum Length of Non-Preemptive Regions Under Fixed Priority Scheduling [*]

Gang Yao, **Giorgio Buttazzo** and **Marko Bertogna**
Scuola Superiore Sant'Anna, Pisa, Italy
{g.yao, g.buttazzo, m.bertogna}@sssup.it

Abstract

The question whether preemptive systems are better than non-preemptive systems has been debated for a long time, but only partial answers have been provided in the real-time literature and still some issues remain open. In fact, each approach has advantages and disadvantages, and no one dominates the other when both predictability and efficiency have to be taken into account in the system design. In particular, limiting preemptions allows increasing program locality, making timing analysis more predictable with respect to the fully preemptive case.

In this paper, we integrate the features of both preemptive and non-preemptive scheduling by considering that each task can switch to non-preemptive mode, at any time, for a bounded interval. Three methods (with different complexity and performance) are presented to calculate the longest non-preemptive interval that can be executed by each task, under fixed priorities, without degrading the schedulability of the task set, with respect to the fully preemptive case. The methods are also compared by simulations to evaluate their effectiveness in reducing the number of preemptions.

1 Introduction

The primary goal of a real-time system is to guarantee the schedulability of the task set on the processing platform so that each task completes its execution within its deadline. Since the pioneering work of Liu and Layland [20], a lot of work has been done in the area of real-time scheduling to analyze and predict the schedulability of the task set under different scheduling policies and several task models. Although the Earliest Deadline First (EDF) algorithm is optimal on a uniprocessor in terms of schedulability [10], other fixed priority schedulers, such as Rate Monotonic (RM) or Deadline Monotonic (DM), are widely used in real-time embedded systems, because they are simpler to implement in current operating systems and are characterized by a lower run-time overhead. A comprehensive comparison of RM against EDF can be found in [9].

The question whether enabling or disabling preemption during task execution has been investigated by many authors under several points of view and it is not trivial to answer. In fact, there are several advantages and disadvantages to be considered when adopting a non-preemptive scheduler, which depend on the particular application requirements to be achieved. In particular, the following issues must be taken into account.

- In many practical situations, such as I/O scheduling or communication in a shared medium, either preemption is impossible or prohibitively expensive.
- Preemption destroys program locality and affects the cache behavior, making the execution times more difficult to characterize and predict [17, 21, 22, 23]. Usually, the preemption overhead is ignored in the scheduling analysis, leading to imprecise results.
- The mutual exclusion problem is trivial in non-preemptive mode, since it naturally guarantees the exclusive access to shared resources.
- In control applications, the input-output delay and jitter are minimized for all tasks when using a non-preemptive scheduling discipline, since the interval between start time and finishing time is always equal to the task computation time [8]. This simplifies the techniques for delay compensation in the control design.
- Non-preemptive execution allows using stack sharing techniques [1] to save memory space in small embedded systems with stringent memory constraints [12].
- A general disadvantage of the non-preemptive discipline is that it may reduce schedulability. In fact, a non-preemptive section of code introduces an additional blocking factor in higher priority tasks that can be taken into account with the same guarantee methods used for resource sharing protocols. In fixed priority systems, however, there are particular cases in which non-preemptive execution improves schedulability (an example is shown below).

[*]This work has been partially supported by the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement no. 216008.

- In non-preemptive systems there is no least upper bound on the processor utilization below which the schedulability of any task set can be guaranteed. This can easily be shown by considering a set of two periodic tasks, τ_1 and τ_2, with priorities $P_1 > P_2$ and utilization $U_i = \epsilon$, arbitrarily small. If $C_2 > T_1$, $C_1 = \epsilon T_1$, and $T_2 = C_2/\epsilon$, the task set is unschedulable, although having an arbitrarily small utilization.

The general problem of finding a feasible schedule with a non-preemptive scheduling algorithm (including those that allow inserted idle time) is intractable (i.e., NP-hard in the strong sense) [16]. When restricting to work conserving algorithms that cannot insert idle times, EDF is still optimal under the non-preemptive case. For concrete fixed priority task systems, preemptive scheduling does not dominate non-preemptive scheduling. Table 1 illustrates a set of periodic tasks (with relative deadlines equal to periods) that can be scheduled under non-preemptive RM, but is unfeasible under preemptive RM[1].

	Comp. time	Period	Arrival time
Task1	2	4	0
Task2	3	6	0

Table 1. A concrete task set that can be scheduled under non-preemptive RM, but not under preemptive RM.

To take advantage of non-preemptive execution without losing the benefits of preemptions, in this paper we consider a limited preemption model, in which a preempted job is not required to surrender the processor immediately. Instead, it may switch to non-preemptive mode and continue to execute for a certain period without being preempted. The maximum length of the non-preemptive interval Q_i is computed to preserve the schedulability of the original (fully preemptive) task set. At runtime, if the remaining execution time of the running job is less than or equal to the maximum length of the non-preemptive region, the job can complete execution without being preempted.

In particular, the addressed problem can be stated as follows:

> Given a set of n preemptive periodic tasks that is feasible under a fixed priority assignment, find the longest non-preemptive region of length Q_i for each task τ_i, so that τ_i can continue to execute for Q_i units of time in non-preemptive mode, without violating the schedulability of the original system.

The algorithm for computing the length of such an interval is implemented off line, so the method does not introduce extra overhead at runtime.

[1]Notice that, under a generic periodic or sporadic model, the task set is unfeasible, as can be seen by delaying the first instance of Task1 by an arbitrarily small instant.

1.1 Related work

Preemption related problems have received considerable attention in the real-time community. Jeffay, Stanat and Martel [16] showed that non-preemptive scheduling of concrete periodic tasks is NP-Hard in the strong sense, and presented a necessary and sufficient condition for the schedulability of non-preemptive periodic tasks under non-idling EDF. A comprehensive schedulability analysis of non-preemptive systems has been done by George, Rivierre and Spuri [14], however, the authors assumed that each task is either completely non-preemptive or fully preemptive.

Fixed priority with deferred preemption has been proposed as a viable alternative by Burns [7]. According to this model, preemption can only occur at some given points in the task code (preemption points), so it is "deferred" when occurring before those points. In this way, each job can be split into several chunks, equal to the number of preemption points plus one. This model represents a flexible trade off between the two extreme cases, which can be controlled by the number of preemption points inserted in the code. The author analyzed this model using the response time approach but did not address the problem of computing the longest non-preemptive regions that keep the task set schedulable. Bril et al. [6] corrected the response time analysis showing some critical situations that may occur in the presence of non-preemptive regions. In particular, the authors illustrated that the largest response time does not necessarily occur in the first job, after the critical instant. Lehoczky [18] addressed the same problem in the fully preemptive fixed priority scheduling when relative deadlines are larger than their respective periods, and introduced the level-i busy period to perform the analysis. The same situation may occur when tasks are scheduled with varying execution priority [15].

A different approach for limiting preemptions in fixed priority systems, based on the concept of preemption thresholds, was proposed by Wang and Saksena [25]. This method allows a task to disable preemption up to a specified priority, which is called preemption threshold. Each task is assigned a regular priority and a preemption threshold, and it is allowed to preempt only when its priority is higher than the threshold of the preempted task.

Dobrin and Fohler [11] proposed a method to reduce the number of preemptions under fixed priority scheduling. They adopt the idea of reassigning attributes to the task (e.g., swapping the priorities or forcing the jobs to be released simultaneously), and create artifact tasks for the instance to solve the inconsistency. However, this algorithm introduces significant computation complexity and requires a large amount of memory.

Baruah and Chakraborty [3] analyzed the schedulability of the non-preemptive recurring task model and showed that there exists polynomial time approximation algorithms for both preemptive and non-preemptive scheduling. Buttazzo and Cervin [8] used the non-preemptive task model to reduce jitter. Ramaprasad and Mueller [21, 22] considered the effects of preemptions on the worst-case execution time (due to the cache behavior) and evaluated how this affects task response times.

Baruah [2] analyzed the problem of deferred preemptions under dynamic priority scheduling, where tasks are scheduled by EDF. The demand bound function is used to determine the largest non-preemptive regions into which each task can be broken up.

In this paper, we also consider the problem of deferred preemptions with the objective of determining the largest non-preemptive regions within each task. The main difference between [2] and our work is that tasks are scheduled based on fixed priorities and extensive simulations are presented to evaluate the proposed methods.

The rest of the paper is organized as follows. Section 2 presents the system model and the terminology adopted throughout the paper. Section 3 introduces the analysis adopted to derive the results. Section 4 describes an exact method for computing the longest non-preemptive regions and also proposes two approximate approaches for simplifying the computation. Section 5 discusses some possible usages of the achieved results and illustrates a concrete example. Section 6 presents some simulation results aimed at comparing the proposed approaches. Finally, Section 7 states our conclusions and future work.

2 Terminology and assumptions

We consider a set $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n periodic or sporadic tasks that have to be executed on a uniprocessor under fixed priority scheduling. A task $\tau_i, 1 \leq i \leq n$, is defined by a worst-case execution requirement C_i, a relative deadline D_i and a period (or minimum inter-arrival time) T_i between two consecutive releases ($D_i \leq T_i$). Task can be scheduled by any fixed priority assignment and are indexed by a decreasing priority, meaning that τ_1 is the highest priority task.

Tasks can be preempted, but contain a set of non-preemptive regions (NP regions) where preemption is disabled and deferred until the end of the region. We define two kinds of non-preemptive regions models:

- *Floating Non-Preemptive regions*. With this model, there is no information on the position of the NP region inside the task code. The only available information concerns the maximum length that NP regions may have inside each task. This model has been adopted for instance in [2] for EDF scheduling.
- *Fixed Non-Preemptive regions*. With this model, the exact location of each NP region is known, so that the schedulability analysis can potentially take advantage of it, as done in [6, 7].

It is worth noting that the first model is more constraining in terms of schedulability, meaning that a feasible task set with floating NP regions is also feasible when the NP regions are located in fixed positions. In this paper, all the results are derived assuming the floating model, which is more general, since it does not require to assume specific knowledge on the location of the NP regions.

The main objective of this work is to compute for each task the longest (floating) non-preemptive region that preserves the schedulability with respect to the fully preemptive case. The following notation is used throughout the paper:

q_i denotes the duration of the longest non-preemptive region of task τ_i.

Q_i denotes the maximum value of q_i that preserves the feasibility of the task set with respect to the fully preemptive case.

B_i denotes the blocking time of task τ_i due to the non-preemptive regions of lower priority tasks.

U_{tot} denotes the total utilization of the task set, that is, the sum of all tasks utilizations:

$$U_{tot} = \sum_{i=1}^{n} C_i/T_i.$$

3 Schedulability analysis

To determine whether a given task τ_i is schedulable, we first need to identify its critical instant, that is the worst-case activation pattern that leads to the largest possible response time for τ_i. For a set of fully preemptive sporadic tasks scheduled by a fixed priority algorithm, it has been proved [20] that the critical instant of task τ_i occurs when it is synchronously activated with all higher priority tasks, and all jobs are released as soon as possible (according to the minimum interarrival time). Given an interval of length t starting in a critical instant, we define the **request bound function** $\text{RBF}(\tau_i, t)$ as the maximum cumulative execution request that could be generated by jobs of τ_i within the considered interval. In [19], it has been shown that

$$\text{RBF}(\tau_i, t) = \left\lceil \frac{t}{T_i} \right\rceil C_i. \tag{1}$$

The cumulative execution request of a task τ_i and all higher priority tasks over an interval of length t is therefore bounded by:

$$W_i(t) = \sum_{j=1}^{i} \text{RBF}(\tau_j, t). \tag{2}$$

The maximum response time in the fully preemptive case is given by the smallest time t^* for which $W_i(t^*) = t^*$. A necessary and sufficient schedulability test is derived in [19] checking whether for every task τ_i there exists a value $t \leq D_i$ such that $W_i(t) \leq t$. The inequality does not need to be evaluated at every $t \in (0, D_i]$, but only at those values of t at which RBF has a discontinuity, i.e., $\{t \in [C_i, D_i] \mid t =$

$k \cdot T_j,\ k \in \mathbb{N}$ and $\forall T_j,\ j \leq i\}$. Moreover, Bini and Buttazzo further reduced the number of points to be checked to the following set [4]:

$$\mathcal{TS}(\tau_i) \doteq \mathcal{P}_{i-1}(D_i) \tag{3}$$

where $\mathcal{P}_i(t)$ is defined by the following recurrent expression:

$$\begin{cases} \mathcal{P}_0(t) = \{t\} \\ \mathcal{P}_i(t) = \mathcal{P}_{i-1}\left(\left\lfloor \frac{t}{T_i} \right\rfloor T_i\right) \cup \mathcal{P}_{i-1}(t). \end{cases}$$

The above set $\mathcal{TS}(\tau_i)$ is referred to as the **testing set** for task τ_i. The size of $\mathcal{TS}(\tau_i)$ is *pseudo-polynomial* in the parameters of the task set [4]. In the remainder of this paper, $\mathcal{TS}(\tau_i)$ is used to compute the longest NP region for each task.

In the presence of non-preemptive regions, an additional blocking factor B_i must be considered for each task τ_i, equal to the longest NP region belonging to lower priority tasks. Therefore, the maximum blocking time that τ_i may experience is:

$$B_i = \max_{k>i}\{q_k\}. \tag{4}$$

The schedulability analysis in the presence of blocking factors has been extended by Bini and Buttazzo in [4], where Theorem 4 can be restated as follows by considering floating NP regions:

Theorem 1. *A task set $\mathcal{T}$ with floating NP regions is schedulable with a fixed priority algorithm if and only if $\forall \tau_i \in \mathcal{T}$ there exists $t \in \mathcal{TS}(\tau_i)$ such that*

$$W_i(t) + B_i \leq t. \tag{5}$$

Notice that condition (5) is necessary and sufficient for guaranteeing the schedulability when considering floating NP regions, while it becomes only sufficient when the regions are fixed. The result of Theorem 1 can be used to determine the maximum amount of blocking a task τ_i can tolerate without missing any of its deadlines. This amount will be called the *blocking tolerance* of task τ_i and will be denoted with β_i. Thus,

$$\beta_i = \max_{t \in \mathcal{TS}(\tau_i)}\{t - W_i(t)\}. \tag{6}$$

Computing β_i with Equation (6) requires the evaluation of all points in the testing set $\mathcal{TS}(\tau_i)$, and has therefore pseudo-polynomial complexity. Figure 1 illustrates a sample task set, showing how $t - W_3(t)$ varies as a function of time. The maximum value of this function, that is the blocking tolerance β_3, is also shown. Notice that β_3 does not correspond to the last point in $\mathcal{TS}(\tau_3)$ (i.e., the task deadline), but it is achieved at an intermediate point.

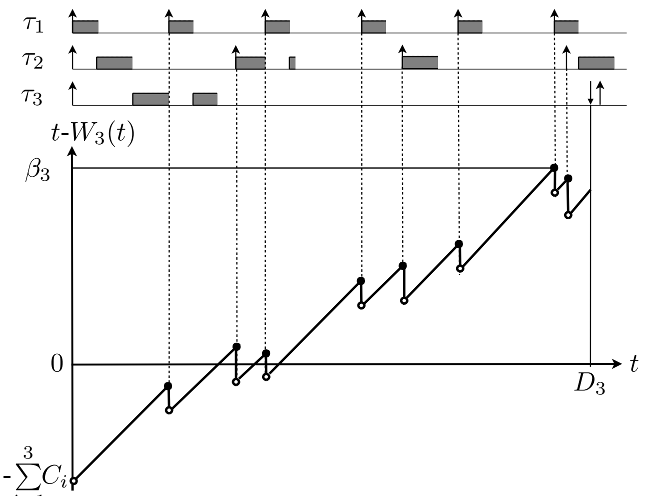


Figure 1. An example for illustrating the blocking tolerance of a task.

4 Longest Non-Preemptive regions

Starting with a fully preemptive task set $\mathcal{T}$, which is schedulable with a fixed priority algorithm, we show how to determine for each task τ_i the largest possible NP region Q_i preserving system schedulability. Being task τ_1 the highest priority task, it cannot be preempted, hence its longest NP region can be arbitrarily large[2] without making the system unschedulable:

$$Q_1 = \infty. \tag{7}$$

The next theorem shows how to derive Q_i for the other tasks.

Theorem 2. *The maximum possible length of any NP region of task $\tau_i, 2 \leq i \leq n$, which guarantees feasibility is given by*

$$Q_i = \min\{Q_{i-1}, \beta_{i-1}\}. \tag{8}$$

Proof. If q_i is the length of the longest NP region of task τ_i, the maximum blocking time that a task τ_k can experience is given by

$$B_k = \max_{i>k}\{q_i\}.$$

Hence, if β_k is the blocking tolerance of τ_k, we must have that:

$$B_k \leq \beta_k$$

that is

$$\max_{i>k}\{q_i\} \leq \beta_k$$

which can be written as

$$\bigwedge_{i>k} (q_i \leq \beta_k)$$

[2]Note that the actual NP region length is limited by the task worst-case execution time, but Q_i is derived without considering such a constraint to obtain a more compact formula.

which is also equivalent to

$$q_i \leq \min_{k<i}\{\beta_k\}$$

Thus, the maximum value for q_i is

$$Q_i = \min_{k<i}\{\beta_k\}. \tag{9}$$

Now, we notice that

$$\min_{k<i}\{\beta_k\} = \min\{\min_{k<i-1}\{\beta_k\}, \beta_{i-1}\}$$

and since

$$Q_{i-1} = \min_{k<i-1}\{\beta_k\}$$

we have that

$$Q_i = \min\{Q_{i-1}, \beta_{i-1}\}$$

which proves Equation (8).

To prove that the Q_i values returned by Equation (8) are also *tight* for each $2 \leq i \leq n$, consider a situation in which τ_i can execute non-preemptively for $(Q_i + \epsilon)$, where ϵ is an arbitrarily small value. Let τ_b be the task with the smallest blocking tolerance among all tasks with higher priority than τ_i. From Equation (9), it follows that $Q_i = \beta_b$. Now, consider the particular arrival sequence in which all tasks $\tau_1, \ldots, \tau_b$ are released synchronously at time t, with all later jobs released as soon as possible, and τ_i starts executing at time $\left(t - \frac{\epsilon}{2}\right)$ in non-preemptive mode for $(Q_i + \epsilon)$ time-units. In this case, τ_b is blocked for $\left(Q_i + \frac{\epsilon}{2}\right) = \left(\beta_b + \frac{\epsilon}{2}\right)$, which is larger than its blocking tolerance β_b. Therefore, τ_b will miss its deadline at time $(t + D_b)$, proving the tightness of Equation (8). $\square$

There can be situations in which the computed Q_i is greater than the worst-case execution time C_i. In these cases, task τ_i can execute in non-preemptive mode for the whole execution time.

4.1 Simplified computation of blocking tolerances

As mentioned in Section 3, computing the blocking tolerances using Equation (6) has a pseudo-polynomial complexity. A simpler but suboptimal solution can be derived by skipping some points in the testing set, so deriving a lower blocking tolerance. Considering that function $t - W_i(t)$ tends to increase with time (as illustrated in Figure 1), we can use the task deadline as a promising testing point for estimating the blocking tolerance:

$$\beta_i^D = \max\{0, D_i - W_i(D_i)\}. \tag{10}$$

For task systems having deadlines equal to periods and scheduled with Rate Monotonic, another simplified method can be found using the utilization test proposed by Liu and Layland in [20]. According to this test, a task τ_i is guaranteed to meet its deadlines if:

$$\sum_{k=1}^{i} \frac{C_k}{T_k} + \frac{B_i}{T_i} \leq U_{lub}(i)$$

where

$$U_{lub}(i) = i(2^{1/i} - 1).$$

Hence, τ_i feasibility is guaranteed under RM if

$$B_i \leq T_i\Big[U_{lub}(i) - \sum_{k=1}^{i} U_k\Big].$$

Thus, using the Liu and Layland feasibility test, the task set is still guaranteed to be schedulable if each task τ_i can tolerate the following blocking time:

$$\beta_i^{LL} = \max\left\{0,\ T_i\Big[U_{lub}(i) - \sum_{k=1}^{i} U_k\Big]\right\}. \tag{11}$$

In Section 6, the pessimism introduced by using such simplified tolerances will be experimentally evaluated with respect to the exact β_i values.

5 Considerations and illustrative example

Once the maximum length Q_i of the NP region has been computed off line for each task, such a result can be used in several ways. Possible options are presented below.

- Exploit NP regions for simplifying the access to shared resources (by encapsulating critical sections into NP regions), so avoiding the implementation of complex concurrency control protocols, like PCP [24] or SRP [1];

- Partition each task into a set of non-preemptive chunks of length no larger than Q_i, by inserting a number of preemption points in the source code in opportune positions, with the objective of reducing the number of preemptions and making the estimation of worst-case execution time more predictable with respect to the fully preemptive case [13];

- Execute a task normally, and switch to non-preemptive mode as soon as a higher priority task arrives. In this way, the preemption can be delayed as much as possible (that is, by Q_i time-units), further reducing the average number of context switches.

- Place a NP region at the end of the task code. In this way, the response time of the task is reduced, since the effect of higher priority jobs arriving at the end of the task execution is postponed until the task completes its execution.

The selection of the strategy to adopt depends on the particular context and is left as a design choice. However, it is worth noting that, when a NP region is located in the last portion of the task code, the analysis can be refined, as done in [6, 7]. In fact, the task response time can potentially be reduced because the higher priority jobs that arrive when τ_i is executing non-preemptively the last chunk of code do not cause interference, allowing for a larger blocking tolerance.

Nevertheless, the analysis turns out to be rather complicated, as shown in [6]. In fact, the worst-case response time of a task τ_i is not necessarily given by the first job of τ_i when it is released synchronously with all other higher priority tasks, because the last NP region of the first job of τ_i might delay the execution of other tasks in a way that causes a larger interference on later jobs of τ_i. We call this phenomenon "*Bril's effect*", from the name of the author who first identified it. Due to this effect, the computation of a tight blocking tolerance for each task is not so straightforward.

5.1 Example

We now present a simple concrete example on a set of four periodic tasks. Relative deadlines are assumed to be equal to periods to compare the effectiveness of all the three proposed methods. Task set parameters, as well as the results obtained from each method, are reported in Table 2. Notice that all three methods use the result of Theorem 2 to compute the longest NP region Q_i of each task, and the only difference is in the way the blocking tolerance is computed. In the following, the `Exact` method refers to the one using Equation (6), `Approx_D` refers to the one using Equation (10), and `Approx_LL` refers to the one using Equation (11).

	Task set		Exact		Approx_D		Approx_LL	
	C	T	β	Q	β^D	Q^D	β^{LL}	Q^{LL}
τ_1	29	85	56	∞	56	∞	56	∞
τ_2	14	92	42	56	20	56	30	56
τ_3	29	127	13	42	12	20	7	30
τ_4	30	925	-	13	-	12	-	7

Table 2. Task set parameters and results.

For the highest priority task τ_1, all three methods produce the same values for Q and β, hence the Q value for τ_2 is also the same. However, the blocking tolerance β_2 is different under the three methods, In particular, notice that the `Exact` method produces the largest blocking tolerance for τ_2 (as expected), whereas the `Approx_LL` method outperforms `Approx_D`. For task τ_3, the situation is different, since `Approx_D` outperforms `Approx_LL`. For task τ_4, β_4 is left empty since τ_4 is the lowest priority task and cannot be blocked by the other tasks in the system.

This simple example shows that the `Exact` method outperforms the other two at the cost of a pseudo-polynomial complexity. Both simplified approaches have $O(n)$ complexity, but no one dominates the other for all the tasks. In the next section, the three methods are better evaluated under different conditions through extensive simulations.

6 Simulations results

In this section, we present some simulation experiments we performed on synthetic task sets to compare the effectiveness of the proposed approaches in reducing the number of preemptions and the length of NP regions. NP regions are assumed to be floating inside the task code, and each task switches to non-preemptive mode for Q_i units of time once a higher priority task arrives.

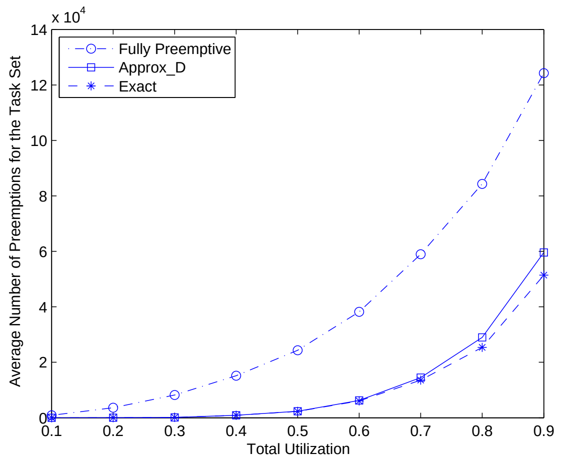
We considered feasible task sets consisting of n periodic tasks with given total utilization U_{tot}. Tasks were synchronously activated at time $t = 0$ and the total simulation time was set to 5 million units of time. For each point in the graph, the result was computed by taking the average over 1000 runs. In particular, the following steps were used to generate a task set:

1. The UUniFast algorithm presented by Bini and Buttazzo [5] was used to generate a set of n tasks with total utilization equal to U_{tot} and individual utilizations U_i uniformly distributed in [0,1].
2. Each computation time C_i was generated as a random integer uniformly distributed in a given interval $[C_{min}, C_{max}]$, and then T_i was computed as $T_i = C_i/U_i$. The reason for generating T_i from C_i is that round-up errors have less influence, since T_i is bigger than C_i, and to avoid generating tasks with $C_i = 1$, which by nature cannot be preempted.
3. The relative deadline D_i was generated as a random integer in the range $[C_i + 0.8 \cdot (T_i - C_i),\ T_i]$, or it was set to T_i when testing the `Approx_LL` method.
4. Each generated task set was verified to be feasible, and unfeasible sets were discarded.

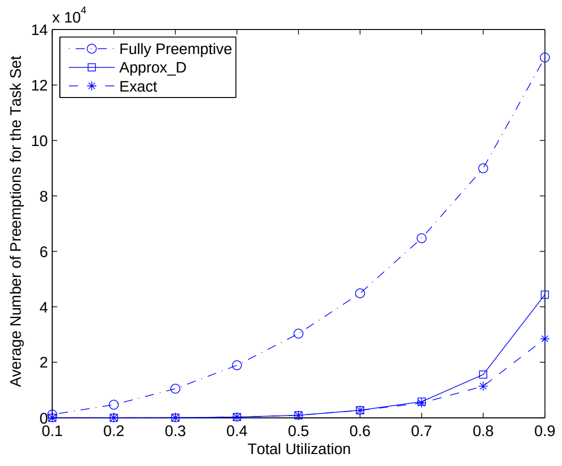
6.1 Experiment 1

In the first experiment, we compared the `Exact` method and `Approx_D` against the fully preemptive case by monitoring the average number of preemptions in all runs (each lasting 5 million units of time), as a function of the total utilization and for different number of tasks. In particular, the total utilization was varied from 0.1 to 0.9 with step 0.1, and the number of tasks was set to 4, 8, 12, and 16, respectively. In this case, each relative deadline D_i was generated as a random integer in the range $[C_i + 0.8 \cdot (T_i - C_i),\ T_i]$. The results are shown in Figure 2.

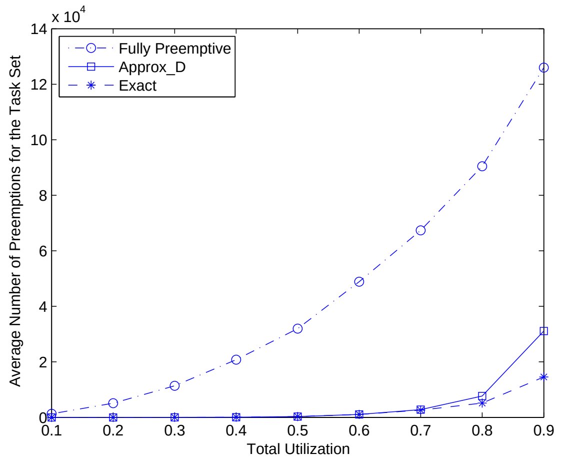
As clear from the graphs, both algorithms are able to reduce the average number of preemptions significantly, with respect to the fully preemptive case, and the advantage of the `Exact` method over `Approx_D` can only be appreciated for total utilizations higher than 0.7. Thus, for workloads smaller than 0.7, it is not worth paying a higher complexity for reducing the number of preemptions. Viceversa,



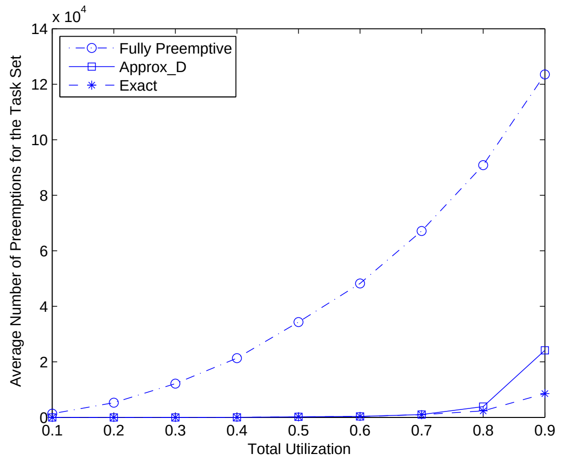
(a) Number of tasks = 4



(b) Number of tasks = 8



(c) Number of tasks = 12



(d) Number of tasks = 16

Figure 2. Average number of preemptions when $D < T$.

for high workloads ($U_{tot} > 0.8$) the higher complexity of the `Exact` method can be justified, since preemptions can still be reduced up to 50% with respect to `Approx_D`.

The reason for having a better performance at low utilizations is that tasks have more slack, thus many of them can execute completely non-preemptively ($Q_i \geq C_i$). Moreover, the performance of both algorithms improves as the number of tasks increases. This can be explained in a similar way, since tasks with low U_i have more slack, resulting in larger NP regions.

The average number of preemptions experienced by each individual task under the different methods was also monitored and it is reported in Figure 3 for $U_{tot} = 0.9$ (in fact, the `Exact` method and `Approx_D` have very close performance at relatively low utilizations).

Notice that high priority tasks can achieve a larger improvement compared with lower priority tasks, as they can reduce the number of preemptions to a larger degree. This can be explained by noting that the $\{Q\}$ sequence is non-increasing, as can be easily seen from Equation (8). Finally, both methods have similar performance for high priority tasks, whereas the `Exact` method can still reduce the number of preemptions up to 50% with respect to `Approx_D`, especially for low priority tasks.

6.2 Experiment 2

In a second experiment, task sets were generated with relative deadlines equal to periods in order to compare the performance of all the three approaches (that is, also including `Approx_LL`). In this case, n was set to 10 and the total utilization U_{tot} was varied from 0.1 to 0.9 (notice that all task sets were verified to be feasible, even for utilizations exceeding the Liu and Layland bound). Simulations with different number of tasks were also performed, but the results are not shown here, because they exhibited a similar performance.

To better illustrate the differences between each curve, Figure 4 reports the ratios of the average number of preemptions with respect to the fully preemptive case. As can be seen, the `Exact` method achieves the best performance, and `Approx_D` outperforms `Approx_LL`, but the differences can only be appreciated for high total utilizations ($U_{tot} > 0.7$).

Figure 5 reports the preemption ratios for each task, when $U_{tot} = 0.7$. Notice that, since τ_1 is never preempted (even in the fully preemptive case), the ratio is not defined for $i = 1$, so the curve starts from $i = 2$. Experiments with lower utilizations were also performed, but they are not reported here, since all the three methods exhibited a similar behavior for every task. Notice how `Approx_LL` deteriorates for low priority tasks ($i > 7$), compared with the other two methods. This can be explained by observing that the Liu and Layland utilization bound for 10 tasks is only slightly higher than 0.7, thus not so much slack can be exploited by `Approx_LL` for the NP regions.

6.3 Experiment 3

In a third experiment, we monitored the length of the NP regions computed by each method under different conditions. The results are illustrated in Figure 6, which plots the average ratio Q_i/C_i for each task and for different workloads. Since Q_1 is set to infinity (see Equation (7)), the curves start from $i = 2$. It is interesting to observe that, for $U_{tot} \leq 0.7$ the average ratio Q_i/C_i produced by all the three methods was greater than one, meaning that, in the average, most tasks executed entirely non preemptively. For $U_{tot} = 0.9$, we can see that most of the higher priority tasks executed non preemptively ($Q_i/C_i \geq 1$), whereas lower priority tasks were preempted to preserve schedulability. However, note that the Q_i/C_i ratio resulting from the `Exact` method is always greater than 0.5, even for the lowest priority task, meaning that schedulability can be preserved with a single preemption point (in the average).

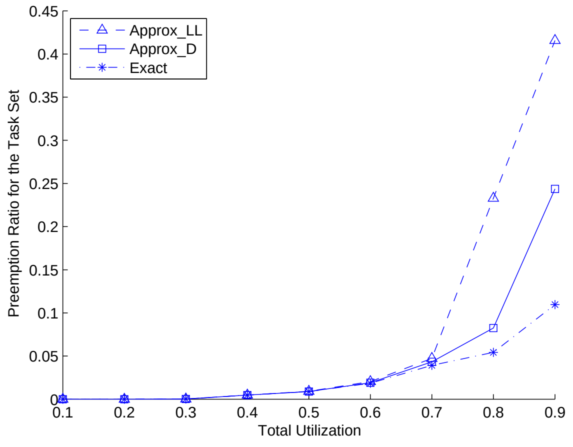


Figure 4. Average preemption ratios when $n = 10$ and $D = T$.

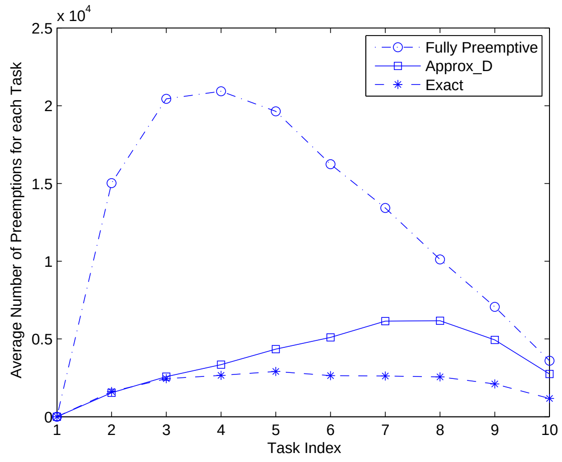


Figure 3. Average number of preemptions for each task when U_{tot} = 0.9 and $D < T$.

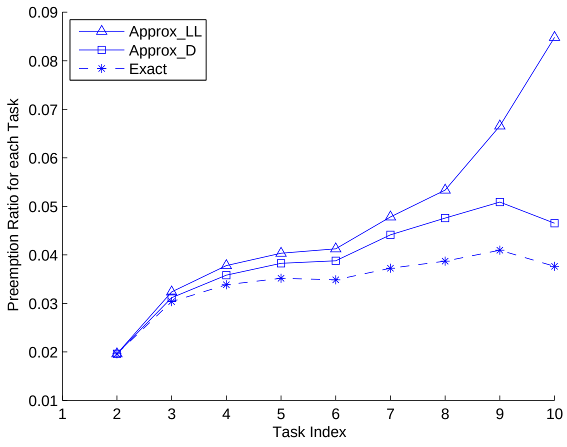


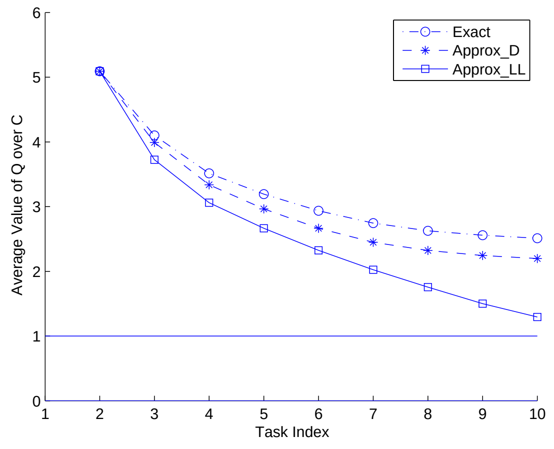
Figure 5. Average preemption ratios for each task when $D = T$ and U_{tot} = 0.7.

6.4 Experiment 4

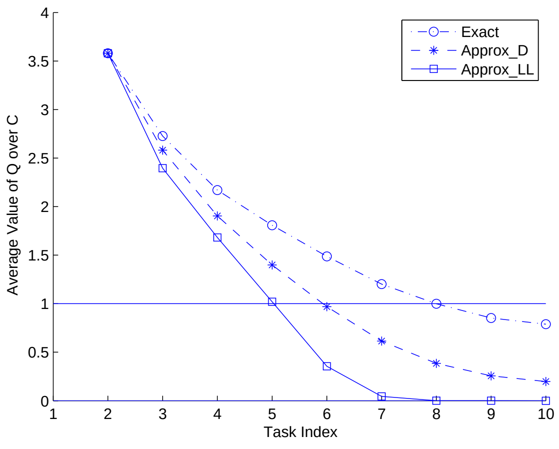
As a final test, we monitored the average number of preemptions for each task instance and for different workloads. Results are reported in Figure 7. As expected, the number of preemptions experienced by a job gradually increases by reducing its priority, and the difference among the three methods is not significant for $U_{tot} \leq 0.7$.

It's worth noting that, when the task set utilization exceeds the Liu and Layland bound, `Approx_LL` starts deteriorating for lower priority tasks, especially for τ_9 and τ_{10}, whose behavior tends to be similar to the one of the fully preemptive case.

For higher utilizations ($U_{tot} = 0.9$), we note that both the `Exact` method and `Approx_D` have similar perfor-

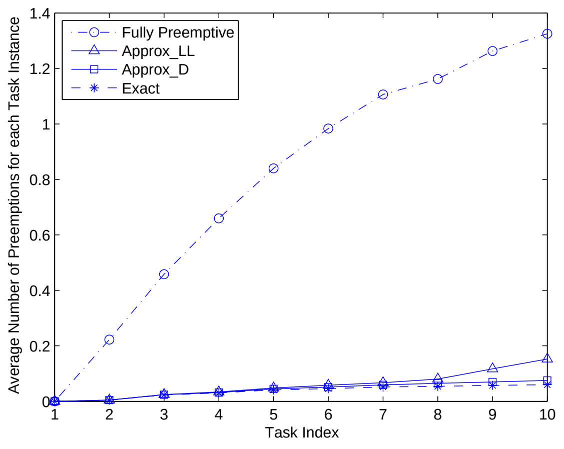


(a) U_{tot}=0.7

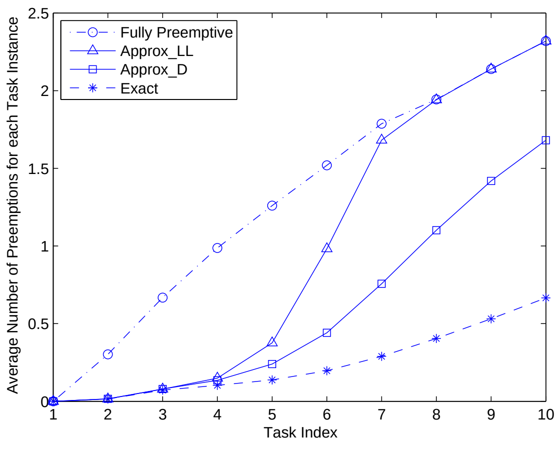


(b) U_{tot}=0.9

Figure 6. Average value of Q over C when $D = T$.



(a) U_{tot} = 0.7



(b) U_{tot} = 0.9

Figure 7. Average number of preemptions for each task instance when $D = T$.

mance for high priority tasks, whereas, for lower priority tasks, the `Exact` method can still achieve more than 50 percent improvement compared to `Approx_D`.

7 Conclusions

In this paper, we considered the problem of limiting the preemptions under fixed priority scheduling, and proposed three methods for computing the longest non-preemptive region within each task that preserves the schedulability of the task set (with respect to the fully preemptive case). The first method, based on exact feasibility analysis, has a pseudo-polynomial complexity and is able to find the maximum possible NP region for each task that preserves schedulability. The other two methods are less precise, since they use two different sufficient feasibility tests to decrease the complexity of the computation. Notice, however, that for static task sets with fixed number of tasks, the longest NP regions can be computed off line, so the higher complexity of the exact method is not payed during runtime.

The contribution of this paper is two-fold. First, we assumed a limited preemption model that integrates the benefit of both non-preemptive and fully-preemptive scheduling to increase predictability without losing schedulability. Second, we proposed three different methods for computing the longest NP region of each task and performed a set of experiments to evaluate the proposed approaches. Simulation results showed that the number of preemptions can be significantly reduced without losing the task set schedulability.

As a future work, we plan to apply the limited preemption model to aperiodic servers and exploit the longest non-preemptive region for providing a better estimate of preemption costs due to cache misses.

References

[1] T. P. Baker. Stack-based scheduling of real-time processes. *Real-Time Systems*, 3(1):67–100, March 1991.

[2] S. Baruah. The limited-preemption uniprocessor scheduling of sporadic systems. *In Proc. 17th ECRTS*, pages 137–144, July 2005.

[3] S. Baruah and S. Chakraborty. Schedulability analysis of non-preemptive recurring real-time tasks. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8 pp.–, April 2006.

[4] E. Bini and G. C. Buttazzo. Schedulability analysis of periodic fixed priority systems. *IEEE Transactions on Computers*, 53(11):1462–1473, 2004.

[5] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Syst.*, 30(1-2):129–154, 2005.

[6] R. J. Bril, J. J. Lukkien, and W. F. J. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited. In *ECRTS '07: Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 269–279, 2007.

[7] A. Burns. Preemptive priority based scheduling: An appropriate engineering approach. *In S. Son, editor, Advances in Real-Time Systems*, pages 225–248, 1994.

[8] G. Buttazzo and A. Cervin. Comparative assessment and evaluation of jitter control methods. *Proc. of the 15th International Conference on Real-Time and Network Systems (RTNS2007)*, pages 137–144, March 29-30, 2007.

[9] G. C. Buttazzo. Rate monotonic vs. edf: judgment day. *Real-Time Syst.*, 29(1):5–26, 2005.

[10] M. Dertouzos. Control robotics: the procedural control of physical processes. *Information Processing*, 74, 1974.

[11] R. Dobrin and G. Fohler. Reducing the number of preemptions in fixed priority scheduling. *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 144–152, July 2004.

[12] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo. A new kernel approach for modular real-time systems development. *13th IEEE Euromicro Conference on Real-Time Systems (ECRTS 2001)*, June 2001.

[13] G. Gebhard and S. Altmeyer. Optimal task placement to improve cache performance. In *Proceedings of the 7th ACM-IEEE International Conference on Embedded Software (EMSOFT 07)*, pages 166–171, Salzburg, Austria, 2007.

[14] L. George, N. Rivierre, and M. Spuri. Preemptive and non-preemptive real-time uniprocessor scheduling. Research Report RR-2966, INRIA, 1996.

[15] M. Harbour, M. Klein, and J. Lehoczky. Fixed priority scheduling periodic tasks with varying execution priority. *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 116–128, Dec 1991.

[16] K. Jeffay, D. Stanat, and C. Martel. On non-preemptive scheduling of period and sporadic tasks. *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 129–139, Dec 1991.

[17] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput.*, 47(6):700–713, 1998.

[18] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 201–209, Dec 1990.

[19] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the Real-Time Systems Symposium - 1989*, pages 166–171, Santa Monica, California, USA, Dec. 1989. IEEE Computer Society Press.

[20] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[21] H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemption points. *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, pages 212–224, Dec. 2006.

[22] H. Ramaprasad and F. Mueller. Bounding worst-case response time for tasks with non-preemptive regions. *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, pages 58–67, April 2008.

[23] H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemptions. *ACM Transactions on Embedded Computing Systems*, 2009. To appear.

[24] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[25] Y. Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on*, pages 328–335, 1999.