Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2010), Macau, China, August 23-25, 2010.

Feasibility Analysis under Fixed Priority Scheduling with Fixed Preemption Points

Gang Yao, Giorgio Buttazzo and Marko Bertogna

Scuola Superiore Sant'Anna, Pisa, Italy, {g.yao, g.buttazzo, m.bertogna}@sssup.it

Abstract

Limited preemption models have been proposed as a viable alternative between the two extreme cases of fully preemptive and non-preemptive scheduling. In particular, allowing preemption to occur only at predefined preemption points reduces context switch costs, simplifies the access to shared resources, and allows more predictable estimations of worst-case execution times. Current results related to such a model, however, exhibit two major deficiencies: (i) The exact response time analysis has a high computational complexity; (ii) The maximum lengths of the non-preemptive regions was not completely investigated in all possible scenarios.

In this paper, we address the problem of scheduling a set of real-time tasks having fixed priorities and fixed preemption points. In particular, under specific but not restrictive assumptions we simplified the feasibility analysis and proposed an efficient feasibility test. Finally, an algorithm for computing the maximum length of fixed non-preemptive regions for each task is described, and some simulation experiments are presented to validate the proposed approach.

I. Introduction

Since the pioneering work of Liu and Layland [20], a lot of research has been done in the area of real-time scheduling to analyze and predict the schedulability of a task set under different scheduling policies and task models. Most of the available results have been derived under a fully preemptive model, where every task can be suspended in any point and at any time, in favor of a task with higher priority. When context switch overhead is ignored in the analysis, as done in most scheduling papers, the fully preemptive model is more efficient in terms of processor utilization, and allows better schedulability results.

In practice, however, arbitrary preemptions can introduce a significant runtime overhead and may cause high fluctuations in task execution times, so degrading system predictability. In particular, three different types of costs need to be taken into account at each preemption [13]. A scheduling cost is due to the time taken by the scheduling algorithm to suspend the running task, insert it into the ready queue, switch the context, and dispatch the new incoming task. A Pipeline cost is due to the time taken to flush the processor pipeline when the task is interrupted and the time taken to refill the pipeline when the task is resumed. A cache-related cost is due to the time taken to reload the cache lines evicted by the preempting task. This time depends on the specific point in which preemption occurs and on the number of preemptions experienced by the task [1], [13].

Moreover, to avoid unbounded priority inversion when accessing shared resources, preemptive scheduling requires the implementation of specific concurrency control protocols, such as Priority Inheritance, Priority Ceiling [24] or Stack Resource Policy [2], which introduce additional overhead and complexity, whereas non-preemptive scheduling automatically prevents unbounded priority inversion.

On the other hand, fully non-preemptive scheduling is too inflexible for certain applications and could introduce large blocking times that would prevent guaranteeing the schedulability of the task set.

To overcome such difficulties, different scheduling approaches have been proposed in the literature to avoid arbitrary preemptions and limit the length of non-preemptive execution.

1) *Fixed Preemption Points (FPP)*. According to this model, each task is divided into a number of non-preemptive chunks (also called subjobs) by inserting predefined preemption points in the task code. If a higher priority task arrives between two preemption points of the running task, preemption is deferred

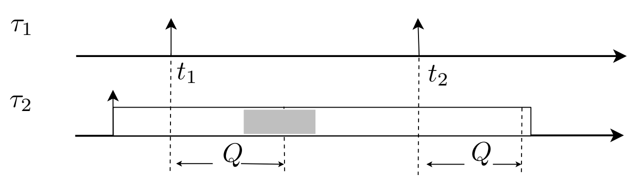
This work has been partially supported by the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement no. 216008.

Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2010), Macau, China, August 23-25, 2010.

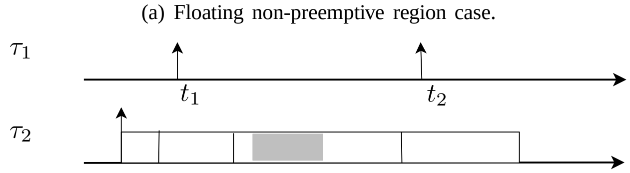
until the next preemption point.

2) *Floating Non-Preemptive Regions (NPR).* Another approach is to define for each task τ_i a maximum interval Q_i in which the task can execute non-preemptively. Since the mode switching is triggered by the arrival time of higher priority tasks, which is unknown a priori, in this model, the non-preemptive regions have no fixed start time, and are considered to be "floating" in the task code.
3) *Preemption Thresholds.* A different approach for limiting preemptions is based on the concept of preemption thresholds, proposed by Wang and Saksena [27] under fixed priority systems. 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 the preemption is allowed to take place only when the priority of arriving task is higher than the threshold of the running task. This work has been later improved by Regehr in [23].

From a practical point of view, using fixed preemption points allows achieving higher predictability. In fact, by properly selecting the preemption points in the code, it is possible to reduce cache misses and context switch costs, therefore improving the estimation of preemption overhead and worst-case execution times [13].



(a) Floating non-preemptive region case.



(b) Fixed preemption point case.

Fig. 1. Floating NPR model vs. FPP model.

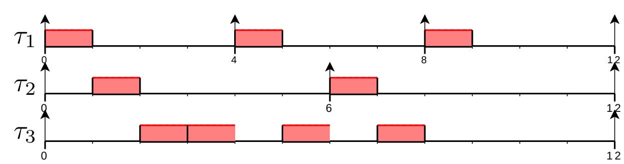
a) Motivating example 1.: To better explain the difference between the floating non-preemptive region and the FPP model, let us consider a simple task set scheduled by these two policies, as depicted in Figure 1. Tasks are assigned fixed priorities and τ_2 has the lowest priority. The gray part inside τ_2 represents a special chunk of code in which a preemption would generate a high preemption cost. Suppose there are two instances of τ_1 arriving at time t_1 and t_2, respectively.

Under the floating case (Figure 1(a)), when τ_1 arrives at time t_1, τ_2 will not be preempted immediately, but will switch to non-preemptive mode and continue for Q units of time. Hence, the first preemption will take place during the execution of the special chunk. For the same reason, the second preemption will take place at time t_2+Q, very close to the end of τ_2, leaving the final non-preemptive region arbitrary small.

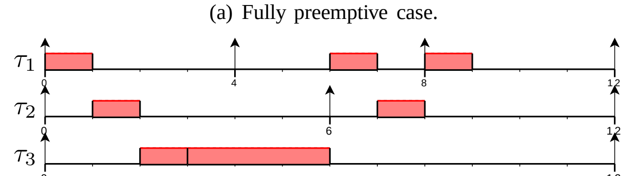
On the other hand, under the FPP case (see Figure 1(b)), τ_2 is divided into four non-preemptive regions and the preemptions are only allowed at these three preemption points. As showed in the figure, the special code chunk can be incorporated into the third non-preemptive region, thus it will never be preempted during its execution. Moreover, the final non-preemptive region of τ_2 cannot be arbitrary small, but has a fixed length decided at design time. For this reason, the second job of τ_1 (arriving at t_2) cannot preempt τ_2.

For the reasons explained above, in this paper we consider a limited preemption model with fixed preemption points (FPP). In this model, the length of the final non-preemptive chunk plays a crucial role in reducing the task response time. In fact, all higher priority jobs arriving during the execution of the final chunk of the running task do not cause a preemption, and their execution is postponed at the end of the task.

b) Motivating example 2.: Let us consider a task set consisting of 3 periodic tasks, with relative deadlines equal to periods. The task set is described as $\mathcal{T} = \{\tau_1, \tau_2, \tau_3\} = \{(1,4),(1,6),(4,12)\}$, where the first number represents the task computation time and the second the period.



(a) Fully preemptive case.



(b) Fixed preemption case: with final subjob long 3.

Fig. 2. Fully preemptive vs. FPP.

Assuming a synchronous activation of the task set, the schedule produced by Rate Monotonic in fully preemptive mode is shown in Figure 2(a). As clear from the figure, τ_3 is preempted twice and has a response time equal to 8 units of time. However, if the last 3 units of τ_3 are executed non preemptively, the two preemptions do not take place and the response time reduces to 6, as shown in Figure 2(b). This simple example clearly shows that the last chunk of a task, when executed in non-preemptive mode, can

Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2010), Macau, China, August 23-25, 2010.

significantly reduce the interference from higher priority tasks, thus reducing the task response time. However, a long non-preemptive region can cause large blocking to higher priority tasks, possibly jeopardizing the system feasibility.

c) Contributions of the paper.: This work provides four main contributions. First, we extend the task model by considering the length of the longest and last non-preemptive region in each task, in order to simplify feasibility test of tasks with fixed preemption points. Second, we identify the conditions under which the feasibility check of a fixed-priority task set can be limited only to the first instance of each task (instead of checking multiple instances within a certain period, as proved by Bril et al. [7]). Third, based on this result, we present an efficient test to verify the feasibility of fixed priority tasks with fixed non-preemptive regions, and finally, we present an algorithm for computing a bound on the length of non-preemptive chunks for each task, discussing how such a bound varies as a function of the length of the final subjob.

d) Paper Organization.: The rest of the paper is organized as follows. Section II presents some related work. Section III introduces the new task model and the methodology used in the paper. Section IV determines the conditions under which the response time analysis for the FPP model can be simplified. Section V presents the feasibility test for fixed priority tasks with given subjob division. Section VI illustrates the algorithm for computing the maximum length of subjobs for each task without violating the system feasibility. Section VII reports some simulation results. Finally, Section VIII states our conclusions and future work.

II. Related Work

Most work on non-preemptive scheduling has typically focused on single-job models, where tasks have precedence relations, are invoked only once, and must be completed before a deadline [11], [12]. Non-preemptive tasks were considered in the Spring Kernel [25], where a heuristic algorithm was used to find a feasible schedule or reduce the number of deadline misses.

A more general characterization of periodic tasks has been considered in [16], [19]. In this model, tasks may have a deadline smaller than or equal to the next release time. For this more general model, Mok [21] has shown that the problem of deciding schedulability of a set of periodic tasks with mutually exclusive sections of code is NP-hard.

Jeffay et al. [15] showed that non-preemptive scheduling of concrete periodic tasks[1] is NP-hard in the strong sense. George et al. [14] provided comprehensive feasibility analysis on non-preemptive scheduling, however, the authors assumed either a completely non-preemptive or a fully preemptive model. Davis et al. [10] considered typical applications of non-preemptive fixed priority scheduling on a CAN bus, and presented the analysis to bound worst-case response times of real-time messages.

Fixed priority scheduling with deferred preemptions, allowed only at some predefined points inside the task code, has been proposed and investigated by Burns [8], who however did not address the problem of computing the maximum length of non-preemptive chunks.

Under the floating model, Baruah [3] computed the longest non-preemptive interval for each task that dose not jeopardize the schedulability of the task set under EDF, with respect to the fully preemptive case. Yao et al. [28] addressed the same problem, but under fixed priorities.

Bril et al. [7] further improved the response time analysis under this model. The authors identified a critical situation that may occur in the presence of non-preemptive regions, deriving the analysis to take such a phenomenon into account. In particular, in certain situations, the execution of the last non-preemptive chunk of a task τ_i can delay the execution of one or some higher priority tasks, which can later interfere with the subsequent invocations of τ_i. Identifying such a situation, later referred to as *self-pushing* phenomenon, requires a more complex test, since the analysis cannot be limited to the first job of each task, but it must be performed on multiple task instances within a certain period. Furthermore, their work dose not address the problem of how to compute the maximum length of each chunk.

When taking preemption costs into account, the schedulability analysis becomes rather complex, because cache-related preemption delays (CRPDs) significantly increase worst-case execution times [17], [26], which in turn affect the total number of preemptions [22]. Under the FPP model, however, the negative influence of CRPDs can be alleviated by appropriately selecting the potential preemption points, and the total number of preemptions a task can suffer is bounded by the number of preemption points.

The research presented in this paper is motivated by the need of limiting both the number and the position of preemptions to better estimate the preemption overhead, reduce the worst-case execution times, and improve the system design. Compared to previous related results [3], [28], this work assumes fixed preemption points instead of arbitrary positions (as illustrated in Figure 1), which allows enhancing the schedulability analysis. Moreover, it provides a method for computing the maximum length of non-preemptive regions. However, the exact estimation of preemption cost is *not* within the scope of this paper, and will be investigated in a future work.

[1] A concrete periodic task is a periodic task that comes with an assigned initial activation.

III. Task Model and Methodology

In this section, we present the task model and the terminology used throughout the paper.

A. Task model

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. Each task τ_i is characterized by a worst-case execution time (WCET) C_i, a relative deadline D_i, and a period (or minimum inter-arrival time) T_i between two consecutive releases. Each task consists of an infinite sequence of jobs $\tau_{i,k}$ $(k = 1, 2, \ldots)$ with arrival time $r_{i,k}$ and absolute deadline $d_{i,k} = r_{i,k} + D_i$. Tasks can be scheduled by any fixed-priority assignment and are indexed by decreasing priority, meaning that τ_1 is the highest priority task. In particular, the following notation is used in the paper:

$$\begin{cases} hp(i) = \{\tau_j |\ j < i\} \\ hep(i) = \{\tau_j |\ j \leq i\} \\ lp(i) = \{\tau_j |\ j > i\} \end{cases}$$

We assume that every task τ_i consists of m_i non-preemptive chunks (subjobs), obtained by inserting $m_i - 1$ preemption points in the code. Thus, preemptions can only occur at the subjobs boundaries. The k^{th} subjob has a worst-case execution time $q_{i,k}$, hence $C_i = \sum_{k=1}^{m_i} q_{i,k}$. In particular, the last subjob of job $\tau_{i,k}$ is denoted as $F_{i,k}$.

To simplify the schedulability analysis, two additional parameters q_i^{max} and q_i^{last} are introduced in the task model:

$$\begin{cases} q_i^{max} = \max_{k=1}^{m_i}\{q_{i,k}\} \\ q_i^{last} = q_{i,m_i} \end{cases} \tag{1}$$

The reasons for choosing these two values can be summarized as follows:

1) Non-preemptive execution can possibly cause blocking to higher priority tasks and the feasibility of a task τ_k is affected by the size q_i^{max} of the longest subjob of each lower priority task $\tau_i \in lp(k)$.
2) For task τ_i, the length q_i^{last} of the final subjob directly affects its response time. In fact, all higher priority jobs arriving during the execution of τ_i's final subjob do not cause a preemption, since their execution is postponed at the end of τ_i (see the examples in Figures 1(b) and 2(b)).

Therefore, we consider each task to be characterized by the following 5-tuple:

$$\{C_i, D_i, T_i, q_i^{last}, q_i^{max}\}.$$

The advantage of such a model will be shown throughout the paper. In the following, the superscript P and FPP will be used to denote that a specific parameter or function refers to the preemptive and FPP model, respectively. In this paper, any time value t is assumed to be a non-negative integer value representing the interval $[t, t+1)$. Tasks may access shared resources, provided that each critical section is confined within one subjob. Preemption cost is ignored in the schedulability analysis, however, it is worth pointing out that by appropriately selecting the preemption points, preemption cost can be reduced and estimated with higher precision compared to arbitrary preemptions.

B. Critical instant

The feasibility check to determine whether a given task τ_i is schedulable under a certain scheduling policy is done under the worst-case scenario that leads to the largest possible response time. The activation times of the tasks causing the worst-case response time of τ_i is defined as the critical instant for τ_i [20].

When tasks have non-preemptive regions, Bril [6] showed that the critical instant of τ_i occurs when it is released simultaneously with all higher priority tasks, and the longest non-preemptive subjob of lower priority tasks starts an infinitesimal time before the release of τ_i.

Bril et al. [7] also showed that, when tasks have non-preemptive regions at the end of their code, the worst-case response time may not occur in the first job. Hence, the feasibility of a task set cannot be checked by analyzing only the first job of each task, as done in fully preemptive systems, but it must be checked for multiple jobs within a certain time interval, which introduces significant computation complexity.

C. Request bound function

Schedulability analysis is performed using the *request bound function* $\text{RBF}(\tau_i, t)$, defined as the maximum cumulative execution request that can be generated by jobs of τ_i within an interval of length t from the critical instant. In [18], it has been shown that

$$\text{RBF}(\tau_i, t) = \left\lceil \frac{t}{T_i} \right\rceil C_i. \tag{2}$$

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

$$W_i(t) = C_i + \sum_{\tau_j \in hp(i)} \text{RBF}(\tau_j, t). \tag{3}$$

A necessary and sufficient schedulability test for fixed priority preemptive tasks was derived by Lehoczky et al. [18], by checking whether for every task τ_i there exists a value $t \leq D_i$ such that $W_i(t) \leq t$. This is stated in the following lemma [18].

Lemma 1. *A fixed-priority task set is feasible under fully preemptive scheduling if and only if* $\forall \tau_i \in \mathcal{T}, \exists t \leq D_i$, *such that*

$$W_i(t) \leq t. \tag{4}$$

where $W_i(t)$ is defined in Equation (3)*.*

If t^* is the smallest value that satisfies Equation (4), then it corresponds to the worst-case response time.

D. Worst-case occupied time

As shown by Bril [7], the worst-case response time of a job can be computed by considering the *worst-case occupied time* $WO_i(C)$, which is the longest possible span of time from the job release till the time at which the job starts or resumes its execution after the completion of C units of computation time. Then, he showed that the worst-case response time WR_i of a task can be expressed in terms of worst-case occupied time WO_i by taking the following limit from the left-hand side:

$$WR_i(C) = \lim_{x \uparrow C} WO_i(x). \tag{5}$$

where $WO_i(x)$ is the smallest $t \in \mathbb{R}^+$ that satisfies

$$t = x \ + \sum_{\tau_j \in hp(i)} \left(\left\lfloor \frac{t}{T_j} \right\rfloor + 1 \right) C_j. \tag{6}$$

Notice that, in Equation (6), the only difference with respect to the worst-case response time is that the ceiling function is replaced by the floor plus one. This essential difference indicates that the response time is computed when the job finishes its execution, regardless of whether other higher priority tasks are released at the end, whereas the occupied time also accounts for the higher priority jobs arriving at the end of the current job's execution.

For example, in the schedule illustrated in Figure 2, the worst-case response time of τ_3 is 8 in Figure 2(a) and 6 in Figure 2(b), whereas its worst-case occupied time is 9 in both cases.

IV. Simplifying Conditions

In this section, we prove that, under the FPP model, the feasibility test can be restricted to the first job of each task, activated at its critical instant, if the following conditions hold:

A1. (Constrained deadlines) $D_i \leq T_i$.

A2. (Preemptive feasibility) The task set is feasible under a fully preemptive model.

Notice that these conditions are not restrictive and are verified for most real-time applications. Burns and Wellings also recognize their relevance in the analysis of non-preemptive tasks [9], although not formally used to derive the results. In this paper, we formally prove that conditions A1 and A2 allow to simplify the feasibility test by restricting the analysis to the first job of each task under the critical instant. We first introduce the concept of *Self-Pushing* phenomenon and derive a number of properties under such a condition, then we prove the main theorem.

A. Properties of the self-pushing scenario

Definition 1. *Under fixed-priority scheduling, a* self-pushing *phenomenon on a task τ_i is defined as the condition in which there exists a job $\tau_{i,k}$, with $k > 1$, such that its response time is larger than the first job under the critical instant, that is:*

$$\exists k > 1, \quad R_{i,k}^{FPP} > R_{i,1}^{FPP}. \tag{7}$$

Notice that $R_{i,k}^{FPP}$ denotes the generic response time of one job while $R_{i,1}^{FPP}$ is the one under critical instance. Now, assume that there exists a self-pushing phenomenon in task τ_i and let $\tau_{i,k}$, $k > 1$ be the first job such that $R_{i,k}^{FPP} > R_{i,1}^{FPP}$. Let $s_{i,k}$ and $s_{i,k-1}$ be the start times of final subjob $F_{i,k}$ and $F_{i,k-1}$, respectively. Such a scenario is illustrated in Figure 3, where the final subjobs are depicted in gray. The following properties can be derived on time interval $[s_{i,k-1}, s_{i,k}]$.

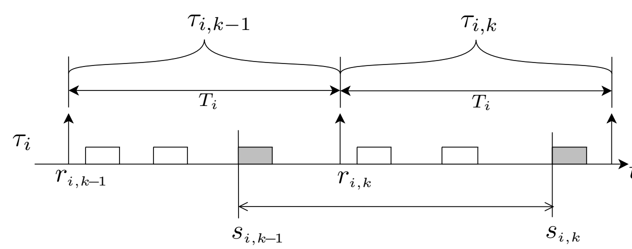


Fig. 3. The self-pushing phenomenon.

Property 1. The start time $s_{i,k-1}$ cannot coincide with the arrival time of tasks from $hp(i)$.

Proof: Since $F_{i,k-1}$ cannot be preempted during its execution, let us consider the start time $s_{i,k-1}$ of $F_{i,k-1}$. If a higher priority job arrives when the final subjob $F_{i,k-1}$ is about to start, then preemption will take place before the execution of $F_{i,k-1}$; that is, $F_{i,k-1}$ will start executing after that higher priority job. Hence, the property holds. ■

Property 2. The interval $[s_{i,k-1}, s_{i,k}]$ is larger than T_i, that is

$$s_{i,k} - s_{i,k-1} > T_i.$$

Proof: According to the definition of self-pushing, we have

$$R_{i,k}^{FPP} = s_{i,k} + q_i^{last} - r_{i,k} > R_{i,1}^{FPP}. \tag{8}$$

Since $\tau_{i,k}$ is the first job experiencing self-pushing, for $\tau_{i,k-1}$ we have

$$R_{i,k-1}^{FPP} = s_{i,k-1} + q_i^{last} - r_{i,k-1} \leq R_{i,1}^{FPP}. \tag{9}$$

Combining Equations (8) and (9), and noticing that $r_{i,k} \geq r_{i,k-1} + T_i$, we have

$$s_{i,k} - s_{i,k-1} > r_{i,k} - r_{i,k-1} \geq T_i$$

which proves the property. ■

Property 3. The processor is always executing jobs from $hep(i)$ in $[s_{i,k-1}, s_{i,k}]$.

Proof: This can be proved by contradiction. Let $t' \in [s_{i,k-1}, s_{i,k}]$ be the first time instant in which the processor is not executing tasks from $hep(i)$. Clearly, t' cannot be in $[s_{i,k-1}, s_{i,k-1} + q_i^{last}]$, since $F_{i,k-1}$ starts executing non-preemptively at $s_{i,k-1}$. Also, since in $[r_{i,k}, s_{i,k}]$ $\tau_{i,k}$ has remaining execution to be completed, t' cannot be in $[r_{i,k}, s_{i,k}]$. Hence, t' must be within $(s_{i,k-1} + q_i^{last}, r_{i,k})$. All tasks from $hp(i)$ arriving before t' must get finished before that time, by definition of t'. If at or after time instant t', some tasks from $hp(i)$ and $lp(i)$ are activated or the processor becomes idle, the overall interference (including blocking) will certainly be no greater than the total delay experienced by the first job (which is activated at the critical instant). Hence, $R_{i,k}^{FPP} \leq R_{i,1}^{FPP}$, which contradicts the self-pushing assumption and proves the property. ■

B. Simplified feasibility analysis

The following lemma uses the previous properties to show that no self-pushing can occur when conditions A1 and A2 are verified.

Lemma 2. *If the task set has constrained deadlines (A1) and is preemptively feasible (A2), then no self-pushing phenomenon can occur under the fixed-priority FPP model.*

Proof: By contradiction. Assume τ_i experiences a self-pushing and let $\tau_{i,k}$ $(k > 1)$ be the first job with $R_{i,k}^{FPP} > R_{i,1}^{FPP}$. We show that this contradicts the preemptive feasibility or the constrained deadline assumption.

Consider a "synthetic" job $\tau_{i,s}^*$, consisting of the final subjob $F_{i,k-1}$ and job $\tau_{i,k}$ excluding its final subjob $F_{i,k}$, i.e., $\tau_{i,s}^* \doteq F_{i,k-1} \cup (\tau_{i,k} - F_{i,k})$. Obviously, $\tau_{i,s}^*$ has the same execution time C_i. Job $\tau_{i,s}^*$ is illustrated in Figure 4. We assume this job arrives at time $s_{i,k-1}$. Since at this time all tasks from $hp(i)$ are finished and subjob $F_{i,k-1}$ can start, the synthetic job will also start upon arrival.

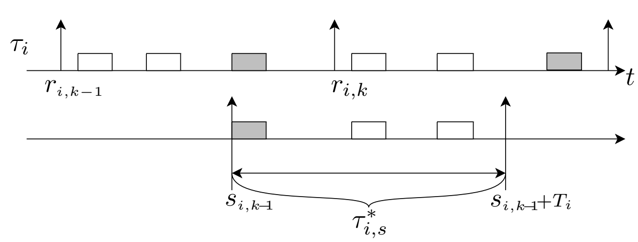


Fig. 4. Synthetic task instance $\tau_{i,s}^*$.

From Property 2, the occupied time of this job, denoted as $O_i^{FPP}(C_i)$, can be expressed:

$$O_i^{FPP}(C_i) = s_{i,k} - s_{i,k-1} > T_i. \tag{10}$$

Under the FPP model, high-priority tasks arriving during the execution of the final subjob are deferred to the end of the running task. Since their start times are aligned with the finish time of the current task, the occupied time under the FPP model takes such interferences into account. And since, from Property 3, in $[s_{i,k-1}, s_{i,k}]$ the processor is executing only tasks from $hep(i)$, job $\tau_{i,s}^*$ suffers no blocking from $lp(i)$. Therefore, the occupied time for this job under P and FPP model will be the same, that is:

$$O_i^P(C_i) = O_i^{FPP}(C_i). \tag{11}$$

Now, from Property 1, we know that $s_{i,k-1}$ cannot coincide with the arrival of tasks from $hp(i)$, hence, the worst-case for job $\tau_{i,s}^*$ is that all tasks from $hp(i)$ arrive at the same time $\epsilon(\epsilon \downarrow o)$ after $s_{i,k-1}$ and function $WO_i^P(x)$ is left-continuous at C_i. Using Equation (5), we have:

$$WR_i^P(C_i) = WO_i^P(C_i) >= O_i^P(C_i). \tag{12}$$

Now, combining Equations (10), (11) and (12) together:

$$WR_i^P(C_i) > T_i.$$

which means that a job with the same parameters as task τ_i will have response time larger than T_i. This contradicts the assumptions and proves the lemma. ■

Using Lemma 2, we can prove the following theorem.

Theorem 1. *Given a preemptively feasible task set with constrained deadlines, the task set is feasible under fixed priority scheduling with FPP, if the first job of each task is feasible under the critical instant.*

Proof: From Lemma 2, we know that there is no self-pushing phenomenon when tasks are preemptively feasible and have constrained deadlines. Hence, for each task τ_i, the response time of any job $\tau_{i,k}$ will be no greater than the one of the first job at the critical instant. That is, $R_{i,k}^{FPP} \leq R_{i,1}^{FPP}$. Hence, if the first job of each task under the critical instant is feasible, then all the forthcoming jobs will also be feasible. The theorem follows. ■

It is worth pointing out that in the proof of Theorem 1 the value of q_i^{last} is never used, meaning that the theorem holds independently of the value q_i^{last}.

V. Feasibility Analysis for the FPP Model

In this section, the result stated in Theorem 1 is used to derive a test for checking the feasibility of a set of fixed priority tasks under the FPP model.

Definition 2. *For each task τ_i, the* subjob allowance α_i *is the length of the longest subjob belonging to lower priority tasks in $lp(i)$. That is,*

$$\alpha_i = \max_{\tau_k \in lp(i)} q_k^{max}. \tag{13}$$

where $q_{n+1}^{max} = 0$ for completeness.

Under fixed priority scheduling with FPP, the presence of non-preemptive subjobs causes the following effects:

On one hand, the non-preemptive execution of any subjob may cause a blocking time to higher priority tasks, however, no job will be blocked after it has started and any job can be blocked for at most once by subjobs belonging to lower priority tasks. Therefore, the maximum blocking time that τ_i may experience is:

$$B_i = \lim_{\epsilon \downarrow o} (\alpha_i - \epsilon)^+ \tag{14}$$

where ϵ is an arbitrary small number to guarantee that subjob from $lp(i)$ actually starts before τ_i. The downarrow in the equation denotes the right-hand limit and the notation x^+ stands for $\max\{x, 0\}$, indicating that the blocking time cannot be negative.

On the other hand, since the final subjob cannot be preempted by any other tasks, it will continue to completion once started. Hence, checking the feasibility of a job is equivalent to checking whether the final subjob can start at least q_i^{last} units of time before the deadline.

Taking into account these two effects, the cumulative execution request under the FPP model, denoted as $W_i^{FPP}(t)$, can be represented as:

$$W_i^{FPP}(t) = (C_i - q_i^{last}) + \sum_{\tau_j \in hp(i)} \text{RBF}(\tau_j, t). \tag{15}$$

Notice that the execution request of τ_i's final subjob (q_i^{last}) is excluded in $W_i^{FPP}(t)$. The feasibility condition for the task set using $W_i^{FPP}(t)$ and α_i is stated in the next theorem.

Theorem 2. *A preemptively feasible task set with constrained deadlines and given subjob division is schedulable under fixed priority with FPP, if for each task τ_i there exists $t \in (0, D_i - q_i^{last}]$ such that*

$$W_i^{FPP}(t) + \alpha_i \leq t. \tag{16}$$

where $W_i^{FPP}(t)$ and α_i are defined in Equation (15) *and* (13), *respectively.*

Proof: We first prove the theorem for tasks with $\alpha_i = 0$. If $\alpha_i = 0$, e.g., the lowest priority task τ_n, the blocking time due to lower priority tasks is zero. Since the non-preemptive execution of subjobs will only possibly reduce the interference and the blocking time is always zero, hence the feasibility can be verified as in the fully preemptive case independent of Equation (16).

When $\alpha_i > 0$, let t^* be the earliest time that satisfies Equation (16). Hence, there $\exists t^* \leq D_i - q_i^{last}$ and:

$$W_i^{FPP}(t^*) + \alpha_i = t^*.$$

Using Equation (2) and (15), this can be written as:

$$(C_i + \alpha_i - q_i^{last}) + \sum_{\tau_j \in hp(i)} \left\lceil \frac{t^*}{T_j} \right\rceil C_j = t^*.$$

which is equivalent to:

$$WR_i^P(C_i + \alpha_i - q_i^{last}) = t^*. \tag{17}$$

Since in this proof all WR and WO functions refer to the preemptive model, we omit the P superscript to simplify the notation. The start time of the final subjob of τ_i is given by $WO_i(C_i + B_i - q_i^{last})$, where B_i is the actual blocking time given by Equation (14). Hence, we have:

$$WO_i(C_i+B_i-q_i^{last}) = \lim_{\epsilon \downarrow 0} WO_i(C_i+\alpha_i-\epsilon-q_i^{last}) \tag{18}$$

According to Equation (5), we have:

$$\lim_{\epsilon \downarrow 0} WO_i(C_i+\alpha_i-\epsilon-q_i^{last}) = WR_i(C_i+\alpha_i-q_i^{last}) \tag{19}$$

Combining Equations (17), (18) and (19) together:

$$WO_i(C_i + B_i - q_i^{last}) = t^*.$$

Therefore, the final subjob will start at t^* and finish at $t^* + q_i^{last}$. Since $t^* \leq D_i - q_i^{last}$, the first job of τ_i meets its deadline and, from Theorem 1, we conclude the entire task is feasible under FPP model. Hence the theorem follows. ■

Condition (16) does not need to be evaluated at every $t \in (0, D_i - q_i^{last}]$, but only at those values of t at which RBF has a discontinuity, i.e. $\{t \in (0, D_i - q_i^{last}] \mid t = k \cdot T_j,\ k \in \mathbb{N} \text{ and } \forall T_j,\ \tau_j \in hp(i)\}$. Moreover, similarly to the methods presented in [4], the number of points can be further reduced to the following set:

$$\mathcal{TS}(\tau_i) \doteq \mathcal{P}_{i-1}(D_i - q_i^{last}). \tag{20}$$

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

$$\begin{cases} \mathcal{P}_0(t) = \{t\} \\ \mathcal{P}_i(t) = \mathcal{P}_{i-1}\left(\left\lfloor \frac{t}{T_i} \right\rfloor T_i\right) \cup \mathcal{P}_{i-1}(t) \end{cases} \tag{21}$$

Theorem 2 allows finding the maximum length that subjobs of tasks in $lp(i)$ can have without jeopardizing the feasibility of τ_i. Thus, from Equation (16), the maximum possible value α_i for task , denoted as *blocking tolerance* β_i, results:

$$\beta_i = \max_{t \in \mathcal{TS}(\tau_i)} \{t - W_i^{FPP}(t)\}. \tag{22}$$

Notice that the lowest priority task τ_n will not be blocked by any other tasks in the system, hence it becomes meaningless to calculate β_n. However, we keep this parameter for the reason of completeness.

Corollary 1. *Given a preemptively feasible task set with constrained deadlines and a specific subjob division, the task set is feasible under fixed priority if $\forall \tau_i, i > 1$*

$$q_i^{max} \leq \min_{\tau_j \in hp(i)} \{\beta_j\}. \tag{23}$$

where β_j is given by Equation (22)*.*

Proof: The corollary can simply be proved through Theorem 2 and the definition of subjob allowance. Note that q_1^{max} is not used in the test since τ_1 does not cause blocking to any other task. For $i > 1$, if q_i^{max} satisfies Equation (23), then from the definition of subjob allowance we know that $\alpha_j(\tau_j \in hp(i))$ will not exceed β_j, hence the schedulability is guaranteed by Theorem 2. ■

Notice that the schedulability for each task τ_i itself is verified by checking the value of $q_j^{max}(\tau_j \in lp(i))$, or as the lowest priority task in the system, is automatically guaranteed as the first part of the proof of Theorem 2. Using the value of β_i, we can derive the feasibility condition for each task. The pseudo-code for the feasibility check is presented in Algorithm 1. Line 2 sets the initial value for τ_1. The *for-loop* in Line 3 checks the task feasibility one by one, in decreasing priority order, using the condition in Corollary 1. If the algorithm reaches Line 7, then all the tasks will be feasible and the algorithm returns $true$, otherwise, if there is a task with q_i^{max} exceeding the maximum possible value (Line 4), it returns $false$, meaning that the task set cannot be guaranteed.

Input: $\{D_i, C_i, T_i, q_i^{max}, q_i^{last}\}$ for $\forall \tau_i \in \mathcal{T}$, preemptively feasible and $D_i \leq T_i$.
Output: Feasibility of the task set under FPP

1 **begin**
2 $\quad \beta_1 = D_1 - C_1$
3 $\quad$ **for** $i \leftarrow 2$ **to** n **do**
4 $\quad\quad$ **if** $q_i^{max} > \min_{\tau_j \in hp(i)}\{\beta_j\}$ **then**
5 $\quad\quad\quad$ **return** "$false$"
6 $\quad\quad$ Calculate β_i using q_i^{last} by Equation (22)
7 $\quad$ **return** "$true$"
8 **end**

Algorithm 1: Feasibility test for a given task set under fixed priority with FPP.

VI. Bound of Subjob Length

In this section, we illustrate a method for computing the maximum subjob length for each task under different circumstances and we discuss how this length varies depending on the length of the final subjob.

Let Q_i be the maximum possible length that any subjob belonging to τ_i can have, without jeopardizing the system feasibility under FPP. Notice that q_i^{max} and q_i^{last} represent the actual lengths in the task code for a given subjob division, whereas Q_i is the upper bound for such lengths. Moreover, Q_i is derived without considering the limitation of the worst-case execution time, hence it can be $Q_i > C_i$.

Corollary 1 already provides a bound for the subjob length of τ_i. However, we now derive an efficient way to compute Q_i recursively.

Since task τ_1 does not cause any blocking to other tasks and it does not experience any interference, we set:

$$\begin{cases} Q_1 = \infty \\ \beta_1 = D_1 - C_1. \end{cases} \tag{24}$$

The next lemma shows how to derive Q_i for the remaining tasks in the system.

Lemma 3. *Given a preemptively feasible task set with constrained deadlines, the maximum length of subjob from task $\tau_i, 2 \leq i \leq n$ that guarantees feasibility under FPP is given by*

$$Q_i = \min\{\beta_{i-1}, Q_{i-1}\} \tag{25}$$

where β_{i-1} can be computed by Equation (22) *and the initial value for τ_1 is given in Equation* (24).

Proof: From Corollary 1, the subjobs length of τ_i must satisfy

$$q_i^{max} \leq \min_{\tau_k \in hp(i)}\{\beta_k\}.$$

So the upper bound of the subjob length of τ_i is given by

$$Q_i = \min_{\tau_k \in hp(i)}\{\beta_k\}. \tag{26}$$

Noting that

$$\min_{\tau_k \in hp(i)}\{\beta_k\} = \min\left\{\beta_{i-1}, \min_{\tau_k \in hp(i-1)}\{\beta_k\}\right\}$$

and that $Q_{i-1} = \min_{\tau_k \in hp(i-1)}\{\beta_k\}$, Equation (26) can be rewritten as

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

which proves the lemma. ∎

It is worth pointing out that the value of Q_i for task τ_i only depends on $\beta_k(\tau_k \in hp(i))$, as expressed in Equation (26). According to Equation (15) and (22), the blocking tolerance β_i is a function of q_i^{last}. Therefore, q_i^{last} does not *directly* affect Q_i, but only the value of β_i, which will be used to compute $Q_j(\tau_j \in lp(i))$. Depending on the knowledge we have on the length of the last subjob, we can distinguish three cases:

- *The value of q_i^{last} is not available*. In this case, the guarantee has to be performed in the worst-case scenario in which τ_i can be preempted arbitrarily near the end of its code. This is equivalent of considering $q_i^{last} = \lim_{\epsilon \downarrow 0} \epsilon$, as done in the floating non-preemptive model. In this case, the upper bound on the subjob length will be denoted as Q_i^{float}.
- *The value of q_i^{last} is given as the design parameter*. In this case, the upper bound Q_i^g is performed as described above.
- *The value of q_i^{last} is equal to q_i^{max}*. In this case, the upper bound on the subjob length will be the highest and will be denoted as Q_i^*.

The subjob division is a compromise of several constraints, e.g. the task structure, application context, hence, the preemption points placement is not only a matter of the length of each NPR, but also the preemption cost at this point and other constraints. Chances are that the length of final NPR is not the longest one, and for the concerning of system schedulability, both q_i^{last}, q_i^{max} and other task parameters must be taken into account, using the methods presented above.

The computation of Q_i^* is done in a similar way as presented in Lemma 3, one task at a time in decreasing priority order. The crucial factor now is the value of q_i^{last}, which is set to the maximum possible value (equal to $\min\{C_i, Q_i^*\}$) to compute the blocking tolerance, which will be used to calculate the bound of NPR length of lower priority tasks.

Observation 1. *Given a preemptively feasible task set with constrained deadlines, in the FPP model we have that*

$$Q_i^* \geq Q_i^g \geq Q_i^{float} \geq 0.$$

Proof: This can be proved by considering the length of the final subjob. For the case of Q_i^*, q_i^{last} has the largest possible value. On the contrary, for Q_i^{float}, q_i^{last} is an arbitrary small number, while for Q_i^g, q_i^{last} has an intermediate value between the two cases.

Now, a larger final subjob reduces the interference from higher priority tasks, allowing a larger blocking time from lower priority tasks. Since the maximum subjob length is equal to the minimum blocking tolerance from $hp(i)$, the observation follows. ∎

VII. Simulation Results

This section presents some experimental results performed on synthetic task sets to compare the maximum subjob length and the average number of preemptions under different situations.

The task set parameters used in the simulations were randomly generated as follows: The UUniFast algorithm [5] was used to generate a set of n tasks with total utilization equal to U_{tot}. Each computation time C_i was generated as a random integer uniformly distributed in a given interval $[5, 50]$, and then T_i was computed as $T_i = C_i/U_i$. The relative deadline D_i was generated as a random integer in $[C_i + 0.5 \cdot (T_i - C_i),\ T_i]$ and the unfeasible task sets under fully preemptive mode were discarded. In all the graphs, each plotted point represents the average value over 1000 randomly generated task sets.

A. Exp. 1: different Q length

In a first experiment, we considered a set of 10 tasks, monitoring the maximum subjob length for each task under different circumstances.

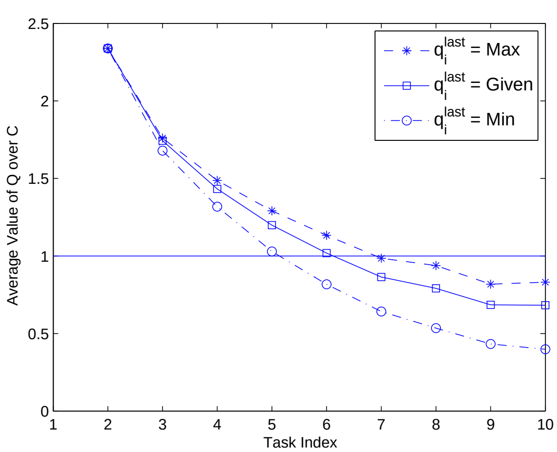


Fig. 5. Average value of Q_i/C_i.

Figure 5 plots the average ratio Q_i/C_i for each task when U_{tot} is equal to 0.9. Simulations were performed under different workloads, however, all the three values resulted to be very similar for low utilizations. Since all three values for τ_1 were set to infinity, the curves start from i=2. The value of Q_i^g was computed by Lemma 3 setting q_i^{last} equal to $\min\{C_i/2, \min_{j<i}\{\beta_j\}\}$.

This result shows that the subjob bound is affected by the length of the final subjob. As expected, Q_i^* is the maximum of all these three values and Q_i^{float} is the smallest. Note that the difference becomes larger for tasks with lower priorities. This is because the lower priority tasks have a larger chance to be preempted by high priority tasks, therefore, the length of the final subjob becomes more crucial: a larger value of q_i^{last} will lead to larger blocking tolerance and consequently larger Q.

B. Exp. 2: average preemption number

In a second experiment, we monitored the average number of preemptions produced in a run (lasting 1 million units of time) as a function of U_{tot}, under different scenarios. Here U_{tot} was varied from 0.5 to 0.95 with step 0.05 and $n = 15$.

Under the floating condition task τ_i switches to non-preemptive mode for Q_i^{float} units of time when a higher priority task arrives [28]. Under the Q_i^* condition, task τ_i executes non-preemptively if $C_i \leq Q_i^*$, otherwise, preemption points are inserted from the end of task code to the beginning, with Q_i^* length interval, i.e., all the subjobs, except the first one, have length equal to Q_i^*. For the sake of comparison, in the case of Q_i^g, we assume preemption points are inserted in the same way as in the case of Q_i^*, but with interval length equal to Q_i^{float} ($Q_i^g = Q_i^{float}$). Figure 6 reports the ratios of average number of preemptions under the different limited preemptive model with respect to the fully preemptive model, as a function of the system utilization U_{tot}.

As clearly showed in the figure, the size of the last subjob is not a crucial parameter for reducing the number of preemptions when the task set utilization is low, whereas its influence becomes more relevant for higher workloads. In this condition, setting q_i^{last} to the maximum value achieves the least number of preemptions.

It is interesting to point out the subtle differences between Q_i^g and Q_i^{float}. Under Q_i^{float} case, each preemption is deferred Q_i^{float} units of time unless the running task remaining execution time is less than Q_i^{float}. While under Q_i^g case, the preemption points are inserted at fixed interval of Q_i^g, hence, each preemption is deferred to the next point and the average deferred time is only around $Q_i^g/2$. Since task computation time is fixed and $Q_i^g = Q_i^{float}$, Q_i^g case should generate more preemptions than the Q_i^{float} case, which is validated through simulation results. A fair comparison can only be done when the preemption cost is also taken into account, which will be a future work.

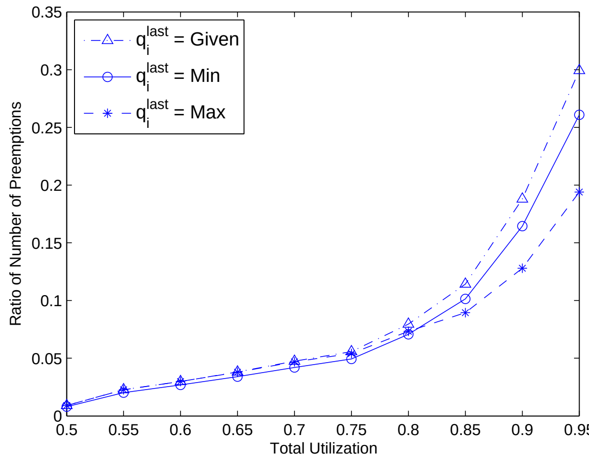


Fig. 6. Ratio of number of preemptions with respect to the fully preemptive case.

VIII. Conclusions

In this paper, we considered the problem of analyzing the feasibility of a task set with fixed preemption points under fixed priority scheduling. The feasibility analysis under limited preemptions has been simplified with respect to the existing literature, proving that, under given conditions, guaranteeing the first job of each task is sufficient for the entire task set. Based on this, an efficient feasibility test under specific but not restrictive assumptions was introduced. We also presented an algorithm for computing the maximum subjob length for each task, and discussed how such a value changes as a function of the final subjob length. Finally, simulations were performed on randomly generated task sets to validate the proposed approach.

As a future work, we plan to exploit the exact preemption position to better estimate the cost of each preemption and task worst-case execution time, thus making the system design more predictable.

References

[1] S. Altmeyer and G. Gebhard. Wcet analysis for preemptive scheduling. In *8th Int. Workshop on Worst-Case Execution Time Analysis*, pages 105–112, Prague, Czech, July 2008.

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

[3] S. Baruah. The limited-preemption uniprocessor scheduling of sporadic systems. In *ECRTS '05: Proc. of Euromicro Conf. on Real-Time Systems*, pages 137–144, July 2005.

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

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

[6] R. Bril. *Specification and Compositional Verification of Real-Time Systems*. PhD thesis, Technische UniversiteitEindhoven (TU/e), 2004.

[7] R. Bril, J. Lukkien, and W. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption. *Real-Time System*, 42(1-3):63–119, 2009.

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

[9] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX (Fourth Edition)*. Addison Wesley Longmain, 2009.

[10] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real-Time System*, 35(3):239–272, 2007.

[11] G. Frederickson. Scheduling unit-time tasks with integer release times and deadlines. *Information Processing Letters*, 16(4):171–173, May 1983.

[12] M. Garey, D. Johnson, B. Simons, and R. Tarjan. Scheduling unit-time tasks with arbitrary release times and deadlines. *SIAM Journal of Computing*, 10(2):256–269, 1981.

[13] G. Gebhard and S. Altmeyer. Optimal task placement to improve cache performance. In *Proc. of the ACM-IEEE Int. Conf. on Embedded Software*, pages 259–268, Salzburg, Austria, 2007.

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

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

[16] E. Lawler and C. Martel. Scheduling periodically occurring tasks on multiple processors. *Information Processing Letters*, 12(1):9–12, 1981.

[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. on Computers*, 47(6):700–713, 1998.

[18] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proc. of the Real-Time Systems Symposium*, pages 166 – 171, CA, USA, Dec 1989.

[19] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

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

[21] A.-L. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD thesis, MIT, USA, 1983.

[22] H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemption points. In *RTSS '06. Proc. of 27th Real-Time Systems Symposium*, pages 212–222, Dec. 2006.

[23] J. Regehr. Scheduling tasks with mixed preemption relations for robustness to timing faults. In *Proc. of the 23rd IEEE Real-Time Systems Symposium*, pages 315–326, 2002.

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

[25] J. A. Stankovic and K. Ramamritham. The spring kernel: A new paradigm for real-time systems. *IEEE Softw.*, 8(3):62–72, 1991.

[26] J. Staschulat and R. Ernst. Multiple process execution in cache related preemption delay analysis. In *Proc. of ACM Int. Conf. on Embedded software*, pages 278–286, Pisa, Italy, 2004.

[27] Y. Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Proc. of Conf. on Embedded and Real-Time Computing Systems and Applications*, pages 328–335, 1999.

[28] G. Yao, G. Buttazzo, and M. Bertogna. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. In *Proc. of Conf. on Embedded and Real-Time Computing Systems and Applications*, pages 351–360, China, 2009.