Limited preemption EDF scheduling of sporadic task systems

Marko Bertogna and Sanjoy Baruah

Abstract—The optimality of the Earliest Deadline First scheduler for uniprocessor systems is one of the main reasons behind the popularity of this algorithm among real-time systems. The ability of fully utilizing the computational power of a processing unit however requires the possibility of preempting a task before its completion. When preemptions are disabled, the *schedulability overhead* could be significant, leading to deadline misses even at system utilizations close to zero. On the other hand, each preemption causes an increase in the *run-time overhead* due to the operations executed during a context switch and the negative cache effects resulting from interleaving tasks' executions. These factors have been often neglected in previous theoretical works, ignoring the cost of preemption in real applications.

A hybrid limited-preemption real-time scheduling algorithm is derived here, that aims to have low run-time overhead while scheduling all systems that can be scheduled by fully preemptive algorithms. This hybrid algorithm permits preemption where necessary for maintaining feasibility, but attempts to avoid unnecessary preemptions during run-time. The positive effects of this approach are not limited to a reduced run-time overhead, but will be extended as well to a simplified handling of shared resources.

Index Terms—Scheduling; preemption; EDF; sporadic tasks; uniprocessors; shared resources.

1 Introduction

It is widely known that preemptive Earliest Deadline First is an optimal scheduling algorithm for single processor systems, meaning that if a task set can be scheduled with a given algorithm, then it can also be preemptively scheduled with EDF[18]. The problem of selecting the appropriate scheduler for a given application scenario could therefore seem definitely solved. However, this result relies on at least two assumptions: (i) a negligible cost of the context changes, and (ii) the independence of the tasks composing the system.

When implementing a preemptive scheduler on a real platform, neither of the above assumptions holds. From one side, preemptions may result in a significant *run-time overhead*, inflating the worst-case execution time of the preempted tasks. This is due to different factors, like the time it takes to store the state of a preempted task to memory, retrieving the context of the preempting one. Also, the cache behavior can be significantly affected by each preemption, resulting in a larger number of cache misses. Since EDF has not been designed to minimize the number of preemptions, it is provably not an optimal algorithm in this case, as will become apparent in this paper.

From the other side, tasks will often need to access shared resources, so that proper protocols for the arbitration of the access to shared resources are needed. The *implementation overhead* of these protocols could be, again, a non-negligible component. The theoretical optimality of protocols like EDF+SRP [1], [3] requires the system to keep track of parameters like ceilings and preemption levels, knowing in advance for how long each task will access each resource. Simpler protocols, like non-preemptive scheduling, are easier to implement but might cause intolerable blocking to safety critical tasks. This is due to the large *schedulability overhead* imposed by non-preemptive scheduling, which refers to the likelihood that any preemptively feasible system will be deemed infeasible due to restrictions that are placed on the scheduling model, i.e., on the possibility of preempting a task with a later deadline[1].

As the discussion above reveals, both preemptive and non-preemptive scheduling have benefits and drawbacks when compared to each other from a pragmatic perspective, and neither model is unequivocally superior to the other.

In this paper, a *limited-preemption* scheduling technique will be presented to join the benefits of both preemptive and non-preemptive scheduling. The run-time behavior of preemptive EDF scheduling will be improved, avoiding preemptions that are not needed to satisfy the schedulability of the system. We will show that it is possible to safely delay the preemption of a task for a specified time interval (given by a proper preemption function), potentially allowing the preempted task to complete its execution. In this way, the theoretical optimality of preemptive EDF is maintained, and a smaller overhead is imposed to the system decreasing the number of context switches. At the same time, the possibility of executing limited sections of code with preemptions disabled will allow an integrated handling of the access to shared resources. Different implementations of the proposed techniques will be discussed, considering the

This research has been supported in part by the National Science Foundation (Grant Nos. ITR-0082866, CCR-0204312, and CCR-0309825).

- *Marko Bertogna is with the Scuola Superiore S.Anna, Pisa, Italy. Sanjoy Baruah is with the University of North Carolina at Chapel Hill. E-mail: m.bertogna@sssup.it, baruah@cs.unc.edu*

1. The discussion refers to EDF scheduling and general uniprocessor feasibility. Within certain scheduling subclasses, like fixed priority scheduling, there are instead sporadic task sets that are not schedulable preemptively (even neglecting the cost of context switches), but that can be scheduled disabling preemptions.

introduced overhead as well as the complexity of the approach.

The remainder of this document is organized as follows. In Section 2, we formally describe the sporadic task model that is adopted in the paper. In Section 3, we provide some background information concerning preemptive and non-preemptive uniprocessor scheduling, along with a description of previously proposed techniques that are based on deferring preemptions. In Section 4, we provide a partial description of our limited-preemption scheduling algorithm, and formally derive some properties that must be satisfied by any sporadic task system that this limited-preemption algorithm *fails* to schedule successfully. By restricting system parameters to ensure that these properties are never satisfied, we are able to complete the design of the limited-preemption scheduling algorithm, in Section 5, in such a manner that feasibility is guaranteed. We illustrate the techniques introduced here by means of a simple 10-task example in Section 6. In Section 7, we address some pragmatic issues that arise in implementing this algorithm. In Section 8, we discuss some simulation experiments that we have conducted to evaluate the effectiveness of our algorithm upon different kinds of workloads. We summarize our results in Section 9.

2 System model

We will consider a set τ of n sporadic tasks [19], [6] that are scheduled upon a single processor platform. A task $\tau_i = (e_i, d_i, p_i)$ is characterized by a *worst-case execution requirement* e_i, a *(relative) deadline* d_i, and a *minimum inter-arrival separation* p_i (which is also sometimes referred to as the *period* of the task). Such a sporadic task generates a potentially infinite sequence of jobs, with successive job-arrivals separated by at least p_i time units. Each job has a worst-case execution requirement equal to e_i and a deadline that occurs d_i time units after its arrival time. All these parameters e_i, p_i, and d_i are real-valued. Jobs are not allowed to suspend themselves.

The *utilization* of task τ_i is defined as $U_i \stackrel{\text{def}}{=} \frac{e_i}{p_i}$. The *system utilization* U of τ is the sum of the utilizations of all the tasks in τ: $U \stackrel{\text{def}}{=} \sum_{i=1}^{n} U_i$. We will denote with $d_{\max}$ (resp. $d_{\min}$) the largest (resp. smallest) relative deadline parameter of any task in τ: $d_{\max} \stackrel{\text{def}}{=} \max_{i=1}^{n}\{d_i\}$ (resp. $d_{\min} \stackrel{\text{def}}{=} \min_{i=1}^{n}\{d_i\}$).

The notion of least common multiple is extended to positive real numbers as follows. Given $a, b \in \mathbb{R}^+$, then

$$\mathsf{lcm}(a,b) = \inf\{x \in \mathbb{R}^+ : \exists p, q \in \mathbb{N}^+ \ x = p\,a = q\,b\}.$$

For a finite set of positive real numbers $\mathcal{A} = \{a_i\}_{i=1}^{m}$, the following recursive definition is adopted:

$$\mathsf{lcm}(\mathcal{A}) = \mathsf{lcm}(a_i, \mathsf{lcm}(\mathcal{A} \setminus \{a_i\})).$$

The *hyperperiod* P of τ will be then defined as $P \stackrel{\text{def}}{=} \mathsf{lcm}(p_1, p_2, \ldots p_n)$.

We consider the scheduling of sporadic task systems upon a single processor, using the **Earliest Deadline First** scheduling algorithm (EDF) [18]. In EDF scheduling, jobs are selected for execution according to their (absolute) deadline parameter, with earlier deadlines favored over later ones (ties may be broken arbitrarily but consistently). In preemptive scheduling, it is assumed that an executing job may be preempted — have its execution interrupted — at an arbitrary instant to be resumed at a later point in time at no additional cost or penalty; in non-preemptive scheduling, by contrast, it is mandated that a job, once it begins execution, is guaranteed exclusive access to the processor until it has completed execution.

A sequence of jobs generated by sporadic task τ_i is said to be *legal* if consecutive jobs arrive at least p_i time units apart. (Observe that a sporadic task can generate infinitely many different legal sequences of jobs, by having different separations between the arrivals of successive jobs.) A *legal collection* of jobs generated by a sporadic task system is defined to be the union of legal sequences of jobs generated by each of the tasks in the system. A sporadic task system is said to be *feasible* if and only if it is possible to schedule all legal collections of jobs that may be generated by this task system in such a manner that all deadlines are met.

In the context of the scheduling of systems of sporadic tasks, a scheduling algorithm is said to be *optimal* if it successfully schedules all legal collections of jobs generated by all feasible sporadic task systems.

For any sporadic task τ_i and any real number $t \geq 0$, the **demand bound function** $\text{DBF}(\tau_i, t)$ is the largest cumulative execution requirement of all jobs that can be generated by τ_i to have both their arrival times and their deadlines within a contiguous interval of length t. It has been shown [6] that the cumulative execution requirement of jobs of τ_i over an interval $[t_o, t_o + t)$ is maximized if one job arrives at the start of the interval — i.e., at time-instant t_o — and subsequent jobs arrive as rapidly as permitted — i.e., at instants $t_o + p_i$, $t_o + 2p_i$, $t_o + 3p_i, \ldots$ Equation (1) below follows directly [6]:

$$\text{DBF}(\tau_i, t) \stackrel{\text{def}}{=} \max\left(0, \left(\left\lfloor \frac{t - d_i}{p_i} \right\rfloor + 1\right) \times e_i\right). \qquad (1)$$

3 Related work

In previous work [19], [6], [16], [14], both *preemptive* and *non-preemptive* scheduling of sporadic task systems was considered.

Under the preemptive paradigm of scheduling, it has been shown [6] that a necessary and sufficient condition for a sporadic task system τ to be EDF-feasible is:

$$\forall\ t \geq 0 \quad \left[\left(\sum_{\tau_i \in \tau} \text{DBF}(\tau_i, t)\right) \leq t\right]. \qquad (2)$$

Since preemptive EDF is an optimal scheduling algorithm [17], the above result represents as well an exact condition for general feasibility.

Non-preemptive EDF scheduling for sporadic task sets has been studied by Jeffay *et al.* in [16], showing that EDF

is optimal even among non-preemptive work-conserving schedulers, i.e., non-preemptive schedulers that do not idle a processor whenever a task is ready to execute. Note that EDF is not optimal among general non-preemptive schedulers (including non work-conserving ones). In the same work, an exact schedulability test with pseudo-polynomial complexity was provided for the case in which deadlines are equal to periods.

Baruah and Chakraborty [4] analyzed the schedulability of non-preemptive task sets under the recurring task model, deriving polynomial time approximation algorithms for both preemptive and non-preemptive scheduling.

The limited preemption model has been introduced by Baruah in [2] as a hybrid policy between fully preemptive and non-preemptive EDF scheduling. The adopted model, upon which our work is based, will be described in Section 4. We are not aware of any other technique used to limit preemptions under EDF. There are instead various approaches applying some kind of limitation in the preemptions under Fixed Priority (FP) scheduling. Wang and Saksena [24] proposed to assign each task a regular priority and a preemption threshold, allowing a task to preempt only when its priority is higher than the threshold of the preempted task. This work has been later improved by Regehr in [21].

Burns presented in [10] a response time analysis for fixed priority task systems under the deferred preemption model. According to this model, each task can be preempted only at pre-defined points, deferring preemption requests that arrive earlier. A similar model has been adopted by Gopalakrishnan *et al.* in [15]. The analysis of these systems has been later improved by Bril *et al.* in [9]. The problem of where to place preemption points in order to reduce the WCET of a task has been analyzed by Simonson *et al.* in [23].

Another limited preemption model has instead been adopted by Yao *et al.* in [26], considering non-preemptive regions that can "float" inside the task code, without relying on fixed preemption point locations. Under this floating model, a method is proposed to compute the size of the largest non preemptive region each task can tolerate without missing any deadline.

4 LIMITED-PREEMPTION EDF SCHEDULING

In limited-preemption EDF scheduling, preemptions are permitted; however, the indiscriminate use of processor preemption is discouraged. Rather, limited-preemption EDF is provided with a *non-preemption function* $Q : \mathbb{R}^{\geq 0} \longrightarrow \mathbb{R}^{\geq 0}$. This function takes as input the time to the deadline of the executing job, and provides the amount of time for which such job could execute non-preemptively. This non-preemption function $Q(t)$ depends upon the parameters of the task system τ being scheduled, and satisfies the property that it is *monotonically non-increasing* in t: for all $t_1 < t_2$, it is the case that $Q(t_1) \geq Q(t_2)$. In Section 5, we will describe how this function is determined for a given τ.

An example of non-preemption function is given in Table 1. Its use will be illustrated via an example in Section 4.1 below.

4.1 Algorithm description

Limited-preemption EDF prioritizes jobs during run-time just as "regular" preemptive EDF does: jobs with earlier (absolute) deadlines are accorded higher priority, with ties broken arbitrarily but in a consistent manner. An executing job is in one of two modes: *regular* or *non-preemptive*. When a job is first selected for execution, it executes in regular mode. Suppose that a job J in regular mode with (absolute) deadline D_o is executing at time-instant t_o, and has e_o units of execution remaining. At this time-instant, a new job with a deadline $< D_o$ arrives. *Job J is not required to surrender the processor immediately; instead, it undergoes a mode-shift to non-preemptive mode*. It may execute in non-preemptive mode for up to $\min(e_o, Q(D_o - t_o))$ additional time-units but must surrender the processor after then.

As an example, let us suppose that limited-preemption EDF is implemented using the non-preemption function Q depicted in Table 1 (Section 6). Suppose that a job with deadline at time-instant 100 is executing in regular mode at time-instant 50, and that it has five additional units of execution remaining. At this time-instant another job, with deadline at time-instant 80, arrives. Since the deadline of the currently-executing job is $(100-50) = 50$ time-units in the future, and $50 \in [10, 60)$, we conclude from the third row of Table 1 that the currently executing task transits to non-preemptive mode, and may execute for up to four additional time-units while in this mode.

4.2 Our result, and its significance

In Section 5 below, we present an algorithm that accepts as input a sporadic task system τ, and efficiently computes a non-preemption function Q, that may be used by limited-preemption EDF. The computed function Q satisfies the following property: *if sporadic task system τ is preemptive uniprocessor* EDF *feasible, then the Q computed by our algorithm, if used with limited-preemption* EDF*, will result in limited-preemption* EDF *always meeting all deadlines.*

We now discuss the significance of this result.

First, observe that as per the claim above, limited-preemption EDF scheduling incurs no **schedulability overhead** over and above the schedulability overhead of fully-preemptive EDF scheduling[2]. In particular, any sporadic task system that is preemptive EDF feasible is guaranteed to also be feasible under limited-preemption EDF. Hence, the benefits described below are achieved by limited-preemption EDF without surrendering the optimality property [18], [13] of fully preemptive EDF.

2. An additional *run-time* overhead might be introduced due to kernel mechanisms required by the algorithm. However, this overhead is often compensated by the reduced number of events the system needs to consider, as described in Section 7.

It has previously been shown [11] that the **total number of preemptions** in any schedule generated by preemptive EDF is bounded from above by at most the number of jobs that are scheduled. However, this result does not in itself bound the number of times that any *individual* job may be preempted — it is quite possible that a large number of jobs are not preempted at all, while an unfortunate few each get preempted a very large number of times. By using limited-preemption EDF, the maximum number of times that any particular task's job is preempted can be bounded as a function of its execution requirement, its relative deadline parameter, and the Q function: since Q is monotonically non-increasing, a "quick and dirty" upper bound on the number of preemptions for a job of τ_i is given by $\lfloor e_i/Q(d_i) \rfloor$.

To better understand how the proposed limited-preemption scheduler could be used to decrease the number of preemptions, consider the following motivating example.

Example 1. *A task system τ is composed by n sporadic tasks $\tau_1, \ldots, \tau_n$. Each task τ_i has a worst-case execution time $e_i = 1$, a relative deadline $d_i = i + 1$ and a period $p_i = n + 1$. The task set is schedulable with preemptive* EDF *according to the test of* (2)*. However, the total number of context switches during the first $n+1$ time units could be as high as $2(n-1)$. This happens when each task τ_i is released at time $(n-i)\varepsilon$, where ε is an arbitrarily small real value. When instead the proposed limited-preemption scheduling model is adopted, the schedulability can be maintained even if each task executes non-preemptively for one time-unit (as will be proved in the next sections). This allows each task to complete its execution before being preempted, significantly decreasing the number of context switches.*

Another important benefit of limited preemption scheduling is related to the case in which tasks can access shared resources. Suppose that a job with absolute deadline D_o is in the midst of executing its **critical section** when a higher priority (i.e., earlier deadline) job arrives at time-instant t_o. If the run-time system determines that this job has less than $Q(D_o - t_o)$ execution units remaining in its critical section, then it can let the job execute through till the end of its critical section before preempting it. Some studies [20], [8] indicate that most critical sections in real-time systems are relatively short. Therefore, considerable savings may be realized for such systems, in terms of both run-time overhead and schedulability overhead, by not needing to repeatedly use non-trivial real-time synchronization protocols to ensure mutual exclusion. Considerations on how to use the non-preemption function for the implementation of simple locking mechanisms will be presented in Section 7. In the following analysis, we will assume that each critical section is fully contained inside a non-preemptive region.

As a last remark, it will be seen (Section 5) that the non-preemption function is computed as a by-product of performing preemptive EDF feasibility analysis; hence, there is no additional **computational effort** needed to determine this function.

4.3 Properties of infeasible systems

In this section, we assume that the specifications of sporadic task system τ, and the non-preemption function Q, are given, and derive some properties that must be satisfied by τ and Q if limited-preemption EDF is to *miss* some deadline while scheduling τ. We will use these properties in Section 5 to design an algorithm for computing Q from τ, such that these properties cannot possibly be satisfied; as a consequence, we can conclude that the system, with the computed non-preemption function, always meets all deadlines.

Suppose that limited-preemption EDF misses deadlines on some legal collections of jobs generated by τ, when using non-preemption function Q. Let $\sigma(\tau)$ denote a smallest such legal collection of jobs, in the sense of being comprised of the fewest number of jobs, upon which limited-preemption EDF will miss some deadlines. Let t_f denote the instant at which a deadline is missed in the limited-preemption EDF schedule of $\sigma(\tau)$. Let $t_a < t_f$ denote the earliest arrival time of any job in $\sigma(\tau)$.

Claim 1. *The processor is never idled over $[t_a, t_f)$ in the* EDF *schedule of $\sigma(\tau)$.*

Proof: Suppose that the processor were to be idled at time-instant t', $t_a < t' < t_f$. Since tasks cannot suspend themselves, it must be the case that no jobs are active — i.e., have arrived but not yet completed execution — at t'. Therefore, we could remove from $\sigma(\tau)$ all jobs arrived before t', and the deadline at time-instant t_f would still be missed. But this contradicts the assertion that $\sigma(\tau)$ is the smallest collection of jobs of τ on which limited-preemption EDF misses a deadline. □

Claim 2. *All jobs in $\sigma(\tau)$, except perhaps the one missing a deadline at t_f, receive* some *execution in the limited-preemption* EDF *schedule of $\sigma(\tau)$.*

Proof: If all jobs not receiving any execution are removed from $\sigma(\tau)$, the remaining collection of jobs would also miss a deadline at t_f. But this contradicts the assertion that $\sigma(\tau)$ is the smallest collection of jobs of τ on which limited-preemption EDF misses a deadline. □

Claim 3. *Suppose that there are no jobs in $\sigma(\tau)$ with deadline $> t_f$. Then*

$$\sum_{i=1}^{n} \text{DBF}(\tau_i, t_f - t_a) > t_f - t_a \ . \tag{3}$$

Proof: By definition of DBF, the sum of the execution requirements of jobs of τ_i over any interval of length $t_f - t_a$ is no larger than $\text{DBF}(\tau_i, t_f - t_a)$, for all τ_i. Since a deadline is missed at time-instant t_f, the sum of the

execution requirements of all jobs in $\sigma(\tau)$ is greater than $t_f - t_a$. The claim follows. □

Claim 4. *Suppose that there are jobs in $\sigma(\tau)$ with deadline $> t_f$. All such jobs must execute for some amount in* non-preemptive mode *during $[t_a, t_f)$, in the limited-preemption* EDF *schedule for $\sigma(\tau)$.*

Proof: This, too, follows from the claimed minimality of $\sigma(\tau)$. Observe that when executing in preemptive mode, a later-deadline job does not delay or otherwise interfere with the execution of an earlier-deadline job under EDF. Suppose that some job with deadline $> t_f$ never executes in non-preemptive mode; it therefore does not interfere with the execution of any job with deadline $\leq t_f$. Consequently, the deadline miss at t_f will occur even if this job is removed from $\sigma(\tau)$. □

Claim 5. *Suppose that there are jobs in $\sigma(\tau)$ with deadline $> t_f$. By Claim 4, these jobs all execute in non-preemptive mode during $[t_a, t_f)$, in the limited-preemption* EDF *schedule for $\sigma(\tau)$. Let $[t_1, t_2]$ denote the latest time-interval $\leq t_f$ during which some such job is executing in non-preemptive mode.*

$$Q(t_f - t_1) + \sum_{i=1}^{n} \text{DBF}(\tau_i, t_f - t_1) > t_f - t_1 \ . \tag{4}$$

Proof: Since a job with deadline $> t_f$ was executing in regular (preemptive) mode at time-instant t_1 under EDF, it must be the case that there were no active jobs at that instant with deadline $\leq t_f$ in the EDF schedule of $\sigma(\tau)$. All the jobs that execute over $[t_2, t_f)$ arrive after t_1, and have deadline $\leq t_f$. By definition of the DBF function, the cumulative execution requirement of all these jobs is at most $\sum_{i=1}^{n} \text{DBF}(\tau_i, t_f - t_1)$; since a deadline is missed, this must exceed the interval length, which is $t_f - t_2$. We therefore have

$$\sum_{i=1}^{n} \text{DBF}(\tau_i, t_f - t_1) > t_f - t_2 \ . \tag{5}$$

Let D_j, $D_j > t_f$, denote the absolute deadline of the job that executes non-preemptively over $[t_1, t_2]$. By the semantics of limited-preemption EDF, it must be the case that

$$t_2 - t_1 \leq Q(D_j - t_1)$$

from which it follows (since $t_f < D_j$, and $Q(t)$ is monotonically non-increasing with t), that

$$t_2 - t_1 \leq Q(t_f - t_1) \ . \tag{6}$$

By adding (5) and (6) above, we have

$$Q(t_f - t_1) + \sum_{i=1}^{n} \text{DBF}(\tau_i, t_f - t_1) > t_f - t_2 + t_2 - t_1 \ ,$$

which is as claimed in (4). □

We can now use the properties derived above to obtain necessary conditions for limited-preemption EDF to *fail* to schedule sporadic task system τ successfully:

Theorem 1. *If a deadline is missed when sporadic task system $\tau = \{\tau_1, \ldots, \tau_n\}$ is scheduled by limited-preemption* EDF *using non-preemption function Q, then at least one of the following conditions is satisfied:*

$$\exists\ t_\phi \ : t_\phi \geq 0 \ : \sum_{i=1}^{n} \text{DBF}(\tau_i, t_\phi) > t_\phi \tag{7}$$

or

$$\exists\ t_\phi \ : t_\phi \geq 0 \ : Q(t_\phi) + \sum_{i=1}^{n} \text{DBF}(\tau_i, t_\phi) > t_\phi. \tag{8}$$

Proof: Recall that we had constructed the collection of jobs $\sigma(\tau)$ considered in Claims 1-5 under the assumption that τ is not schedulable under limited-preemption EDF. In essence, Claims 3 and 5 assert that if τ is not schedulable under limited-preemption EDF, then at least one of the following two holds:

1) There are t_f and t_a, $t_f - t_a > 0$, such that Equation (3) holds. Inequality (7) is obtained from this, by setting t_ϕ to be equal to $t_f - t_a$.
2) There is task τ_j, and time-instants t_f and t_1 with $(t_f - t_1) \leq d_j$, such that Equation (4) holds. Inequality (8) is obtained from this, by setting t_ϕ to be equal to $t_f - t_1$.

□

5 Computing the non-preemption function

Notice that Condition (7) of Theorem 1 is the negation of Condition (2), which is necessary and sufficient for feasibility under purely preemptive EDF. Hence, Condition (7) is exactly the condition for *in*feasibility under preemptive EDF. That is, Theorem 1 is essentially asserting that *any sporadic task system τ, known to be feasible under preemptive* EDF*, is feasible when scheduled using limited-preemption* EDF *with non-preemption function $Q : \mathbb{R} \longrightarrow \mathbb{R}^{\geq 0}$, provided that Condition* (8) *is never satisfied*. In this section, we describe how we can compute function Q for a given sporadic task system τ such that Condition (8) is never satisfied. In computing such a function Q, our objective is to have Q be as large as possible (thereby permitting limited-preemption EDF to execute jobs in non-preemptive mode for longer amounts of time).

It follows from Equation (1) that $\text{DBF}(\tau_i, t)$ does not change for values of t between $(\ell \cdot p_i + d_i)$ and $((\ell+1) \cdot p_i + d_i)$ for each integer $\ell \geq 0$. The non-preemption function Q that we will construct also satisfies this property. As a consequence, Conditions (7) and (8) of Theorem 1 need to be evaluated only at those values of t satisfying $t \equiv (\ell \cdot p_i + d_i)$ for some i, $1 \leq i \leq n$, and some integer $\ell \geq 0$. Let $D_1, D_2, D_3, \ldots$ denote all such t, indexed according to increasing value (i.e., with $D_k < D_{k+1}$ for all k). Then, checking Condition (7) is equivalent to checking the following:

$$\exists\ D_k :: \sum_{i=1}^{n} \text{DBF}(\tau_i, D_k) > D_k \tag{9}$$

and checking Condition (8) is equivalent to checking the following:

$$\exists\ D_k :: Q(D_k) + \sum_{i=1}^{n} \text{DBF}(\tau_i, D_k) > D_k \qquad (10)$$

To ensure that Condition (10) is never satisfied, we would need to ensure that $Q(D_k) \leq (D_k - \sum_{i=1}^{n} \text{DBF}(\tau_i, D_k))$ for all k; this, in conjunction with our requirement that the non-preemption function Q be monotonic non-increasing, suggests the following recursive definition for Q:

$$\begin{aligned} Q(D_1) &= D_1 - \sum_{\tau_i \in \tau} \text{DBF}(\tau_i, D_1) \\ Q(D_k) &= \min\left(Q(D_{k-1}), D_k - \sum_{\tau_i \in \tau} \text{DBF}(\tau_i, D_k)\right) \end{aligned}$$

from which it follows that $Q(D_1), Q(D_2), \ldots$, can be computed iteratively one after the other; a straightforward implementation takes $O(n)$ time per iteration.

Observe that checking Condition (9) is equivalent to finding some D_k such that $Q(D_k) < 0$. Thus, performing preemptive EDF feasibility analysis is reduced to a problem of ensuring that the function $Q(t)$ is non-negative for all $t \geq 0$.

For how many deadlines D_k must we compute $Q(t)$, in order to be able to guarantee that Condition (9) is not satisfied for any t (and we have thus determined the feasibility of τ)?

First, observe that limited-preemption EDF will use computed values of $Q(t)$ only for $t \leq d_{\max}$ during run-time, since no job generated by task system τ will have a deadline greater than $d_{\max}$ time units into the future. Hence if we *a priori* know that τ is feasible when scheduled using fully preemptive EDF, then we can cease computing the Q function once $Q(d_{\max})$ has been determined.

If we do not *a priori* know that τ is feasible under preemptive EDF, then we must ensure that $Q(t)$ is always non-negative — as argued above, this is equivalent to ensuring that τ is feasible under preemptive EDF. *Upper bounds* have been determined such that, if Condition (7) is not violated at some t no larger than these upper bounds, then it can be concluded that the condition will not be violated for any t. This upper bound is [5], [6], [14]

$$T^*(\tau) \stackrel{\text{def}}{=} \min\left[P, \max\left(d_{\max}, \frac{1}{1-U}\sum_{i=1}^{n} U_i \cdot (p_i - d_i)\right)\right] \qquad (11)$$

The bound (11) may in general be exponential in the parameters of τ, since P may be as large as $\prod_{i=1}^{n} p_i$. However, this bound (11) is pseudo-polynomial if the system utilization U is *a priori* bounded from above by a constant less than one. (Any system with system utilization U greater than one is clearly not feasible on a unit-capacity processor; hence, requiring that U be at most c, for some constant $c < 1$, is in effect "wasting" or "writing off" at most a fraction $(1-c)$ of the processor's computing capacity.) It thus follows [6] that *limited-preemption* EDF*-feasibility analysis can be performed in pseudo-polynomial time, for sporadic task systems that have system utilization bounded from above by a constant strictly less than one.*

The algorithm described above for computing the non-preemption function Q for a given sporadic task system τ is presented in pseudo-code form in Figure 1.

Note that, for all $t < d_{\min}$, we artificially set $Q(t) = \infty$. This is possible because no job can be preempted when it is less than $d_{\min}$ time-units from its deadline.

6 An Example

In this section, we illustrate the ideas introduced in this paper by means of a simple example. We consider the sporadic system τ of 10 tasks, depicted in Table 1. The system utilization ($\sum_{i=1}^{10} e_i/p_i$) is equal to 0.93544, and the feasibility interval can be bounded at time 725 (using the bound $T^*(\tau)$ of Equation (11) above).

Since the smallest deadline parameter is 8, it follows that $D_1 = 8$. In line (1) of the pseudo-code presented in Figure 1, $Q(D_1)$ is computed to be equal to $(8-2)$, i.e., 6.

The for loop of lines (2a)-(2c) now executes for $k = 2, 3, \ldots,$ until either $D_k > 725$ or a negative value is computed for $Q(D_k)$. We trace the execution of this loop for selected values of k.

The value of D_2 is 10, and $Q(D_2)$ evaluates to $(10-(2+4)) = 4$. For $k = 3, \ldots, 11$, the corresponding values of D_k are (15, 16, 24, 30, 32, 40, 48, 50, and 56). For each of these values of D_k, the value of $Q(D_k)$ computed in line (2b) remains equal to 4. For $k = 12$, however, $D_{12} = 60$ and $Q(60) = [60 - (7 \cdot 2 + 3 \cdot 4 + 2 \cdot 2 + 4 + 3 + 4 + 8 + 5 + 3)] = (60 - 57) = 3$. D_{13} equals 64, and $Q(64)$, too, evaluates to 3. However at D_{14} (which equals 65), $Q(65)$ as evaluated in line (2b) equals 0.

In order to complete the feasibility test, we need to verify that $Q(D) \geq 0$ for all $D \leq 725$. This turns out to be true (and hence system τ is indeed feasible under preemptive scheduling as well).

Observe that according to the computed values of the non-preemption function Q, tasks τ_1 through τ_6, and task τ_9, may be scheduled entirely non-preemptively, and each job of tasks τ_7 and τ_8 needs to be preempted only once each. We thus see that the power of preemption, while perhaps required in order to ensure feasibility, is necessary for only some of the tasks in the system, and is not required all that often, in this particular example. These results seem pretty typical of other examples we have considered in our experiments. In general, many tasks may be executed entirely non-preemptively (indeed, the task with minimum relative deadline can *always* be executed non-preemptively), and the non-preemption parameter is quite large compared

Input: Sporadic task system $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$; $\tau_i = (e_i, d_i, p_i)$ for all i, $1 \leq i \leq n$.
Let $D_1, D_2, \ldots,$ be the elements of the ordered set $\{d_i + \ell p_i, \forall \ell \in \mathbb{N}, 1 \leq i \leq n\}$ (where $D_k < D_{k+1}, \forall k$).

1) $Q(D_1) \leftarrow D_1 - \sum_{\tau_i \in \tau} \text{DBF}(\tau_i, D_1)$
2) for $k \leftarrow 2, 3, \ldots$ do
   a) if $D_k > T^*(\tau)$ as given by Equation (11) then return "feasible"
   b) $Q(D_k) \leftarrow \min\left(Q(D_{k-1}), D_k - \sum_{\tau_i \in \tau} \text{DBF}(\tau_i, D_k)\right)$
   c) if $Q(D_k) < 0$ then return "infeasible"

Fig. 1. Algorithm for computing the non-preemption function (the function $Q : \mathbb{R} \longrightarrow \mathbb{R}^{\geq 0}$) for a collection of sporadic tasks, while simultaneously checking for preemptive feasibility.

to the execution requirement parameter for very many other tasks.

Let us now revisit the example considered in Section 4.1. Suppose that a job of task τ_7 arrives at time-instant 40, and gets access to the processor for the first time at time-instant 47. Over $[47, 50)$, this job executes in regular mode. Then at time-instant 50, a job of task τ_4 arrives, with a deadline at time-instant $50 + 30 = 80$. The currently executing job of τ_7 has a deadline 50 time units in the future; consequently, it may transit to non-preemptive mode and execute for $Q(50) = 4$ additional time units, before having to surrender the processor to τ_4's job at time-instant 54.

7 Implementation considerations

To take advantage of the potential benefits of limited-preemption scheduling, it is necessary to pay particular attention to some implementation detail, like the run-time mechanisms adopted to enforce the scheduling policy, the memory occupancy of the preemption function, and the locking protocols that can be adopted to arbitrate the access to shared resources. We will hereafter detail such practical considerations, proposing implementations that can improve the run-time behavior of the scheduler.

7.1 Scheduling mechanisms

The implementation of a limited-preemption EDF scheduler requires a mechanism to enforce the maximum time for which an executing task could delay the preemption requests of higher priority jobs. This can be done by setting a timer to the corresponding Q value at the time a preemption is requested. Note that the timer is set only at the first time a higher priority job tries to preempt the executing task. Additional preemption requests from other jobs can be ignored until the timer expires. When this happens, an interrupt is triggered whose only effect is to call the scheduling function. The scheduler will then simply select for execution the job with the earliest deadline, according to the EDF rules.

The operations involved are therefore just the following:

- At the beginning, the earliest deadline task is selected for execution.
- The first time a higher priority job arrives, the time-to-deadline of the executing task is computed as the difference between absolute deadline and current time. The Q value corresponding to the computed time-to-deadline is used to program a timer.
- Further preemption requests arriving before the timer expires are ignored.
- When the timer expires, or a task completes its execution, stop the timer and return to the first point.

The scheduling overhead of the proposed limited preemption algorithm is therefore comparable to the overhead of classic preemptive EDF systems. The only additional penalty introduced by the considered implementation is given by the time spent setting/resetting the timer after the first preemption request. However, the possibility of ignoring later preemption requests until the timer expires, or the job completes its execution, can even lead to a reduction in the overhead. Furthermore, the frequency at which the timer is set is inversely proportional to the values assumed by the Q function. As we will show in our simulations, these values are typically comparable to the worst-case execution times of the tasks, resulting in a very limited number of timer update events.

It is worth noting that typical event-driven implementations of periodic task systems are likely to have one timer per task[3]. Inserting one additional timer dedicated to the handling of non-preemptive sections does not seem a significant increase in the scheduling overhead. Note that the timers involved in the discussion are software timers; only one physical timer is needed, onto which the events signalled by each software timer are queued. The hardware timer is programmed to fire at the earliest such events.

7.2 Memory occupancy

In implementing limited-preemption EDF scheduling, one important pragmatic concern that must be taken into consideration is the storage and retrieval of the non-preemption function. For most sporadic task systems, the non-preemption function Q computed using the algorithm of Section 5 (Figure 1), contains reasonably few

3. In a RTOS, timers are typically used to activate successive instances of a periodic task, or to signal when a deadline is reached.

τ_i	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8	τ_9	τ_{10}
e_i	2	4	2	4	3	4	8	5	3	4
d_i	8	10	15	30	50	50	60	60	60	100
p_i	8	20	25	35	50	90	110	105	100	110

t	$Q(t)$
$[0, 8)$	∞
$[8, 10)$	6
$[10, 60)$	4
$[60, 65)$	3
$[65, \infty)$	0

TABLE 1
Example task system, and the non-preemption function Q computed from it.

distinct entries — the example of Section 6, in which the non-preemption function can be represented as a table with fewer entries than there are tasks in the system, is fairly typical in practice.

In the worst case, however, these tables can be quite large. Notice that the algorithm of Figure 1 may potentially assign a new value to $Q(D_k)$ at line (2b), for each time-instant $D_k \leq d_{\max}$ which is a deadline for some job when all tasks in τ generate a job at time-instant 0 and subsequent jobs as soon as legally permitted to do so. Each task τ_i may therefore contribute as many as $(d_{\max} - d_i)/p_i$ entries to the table; consequently, the tabular representation of the non-preemption function may contain pseudo-polynomially many entries. If stored as a sorted table (as in Table 1), using binary search for table lookup may therefore take time logarithmic in the *values* of the relative deadline parameters. Since such table lookups occur during system run-time, this could be considered unacceptably high. If this be the case, rows of the table may be discarded: doing so would result in limited-preemption EDF making suboptimal decisions regarding how much longer a job that is transiting to non-preemptive mode may execute prior to having to surrender the processor, but there is no danger of deadlines being missed as a result. One possibility is to retain only those entries of the table that correspond to the relative deadline parameters of the tasks in τ — i.e., to only store $Q(d_1)$, $Q(d_2)$, ..., $Q(d_n)$; a job J in regular mode with absolute deadline D_o will then execute in non-preemptive mode for up to $Q(d_i)$ additional time units when a higher priority job arrives at time t_o, being d_i the lowest relative deadline $\geq (D_o - t_o)$.

Note that both the complete and the simplified approaches are superior to the technique adopted in the original paper [2], in which each task τ_k had a fixed non-preemption region length, equal to $q_k = Q(d_k)$, independently from the time to deadline $(D_o - t_o)$ of the considered instance J. Since Q is monotonically non-increasing, and $(D_o - t_o) \leq d_i \leq d_k$, the allowed non-preemptive region length in [2] cannot be larger than in the methods proposed above: $q_k \leq Q(d_i) \leq Q(D_o - t_o)$.

7.3 Access to shared resources

As mentioned in Section 4.2, the possibility of executing sections of code with preemptions disabled can be exploited to implement an efficient mutual exclusion mechanism for the handling of shared resources. To do that, it is sufficient to modify the locking function that is called at the beginning of each critical section to perform the following operations:

- if the executing task is in non-preemptive mode, the remaining time for which it is allowed to execute with preemptions disabled is read from the dedicated timer (as described in Section 7.1);
- if such time is smaller than the critical section length, the task exits from non-preemptive mode and the scheduler is called;
- otherwise, the task simply enters the critical section.

The rationale behind this locking protocol is to execute each critical section in non-preemptive mode, allowing each lock to be released before the timer expires.

In order for the algorithm to work properly, it is necessary that each critical section be smaller than the minimum non-preemptive region of the task that accesses it. As noticed in [20], [8], critical sections are seldom longer than a few instructions. Since Q is a non-increasing function, the minimum non-preemptive region for a task τ_i cannot be lower than $Q(d_i)$. As we will show in our experiments, this value is typically comparable to the task's execution time, and, therefore, much larger than the length of a critical section. If this is not the case, i.e., there are very long critical sections that cannot be avoided, more complex protocols, like the Priority Ceiling Protocol (PCP) [22], [12] or the Stack Resource Policy (SRP) [1], should be used.

The advantage of the proposed protocol over ceiling-based policies such as PCP and SRP is twofold. First, there is no need of complex mechanisms such as those involving the definition of the preemption levels and the computation of the (dynamic) system ceiling. Second, the definition of the (static) resource ceilings requires the system to know in advance which task will access each resource, which is often something difficult to know. Instead, with the protocol described in this section, a task does not need to declare in advance which resource it will access, as long as it will hold each lock for less than $Q(d_i)$ time-units.

The overhead introduced by the proposed limited-preemption locking protocol is given by the timer read operation performed when locking a shared resource, which is typically a fast operation.

7.4 Static preemption points

When evaluating the preemption overhead incurred by a task set scheduled with limited-preemption EDF, a limitation is given by the difficulty of knowing where a task will be preempted. Even if the total number of preemptions is reduced with the proposed algorithm, the net effect on the total worst-case execution time could be limited. This is because a task can still be preempted at an arbitrary point during its execution, potentially causing a large preemption delay.

To sidestep this problem, a possibility is to adopt a static placement of preemption points inside each task's code, similarly to the deferred preemption model adopted for fixed priority systems in [10], [9]. The advantage of this method is that cache-intensive sections of code that repeatedly access the same sets of data can be protected within non-preemptive regions, placing preemption points only at their boundaries. In this way, it is possible to simplify and improve the estimation of the worst-case execution time, significantly reducing the cache related preemption delay, which is the main component of the overhead experienced at each context switch by modern computing systems.

The drawback of adopting static preemption points is that it would be necessary to change the preemption points each time a new task joins the system. This is because the maximum non-preemptive region of a task depends on the parameters of the other tasks. Admitting a new task into the system could decrease the maximum distance allowed between two consecutive preemption points of another task, potentially requiring a re-placement of the preemption points. Therefore, the method described in this section is less suitable for dynamically varying systems.

The analysis developed in the previous sections can be easily applied to the case with fixed preemption points. The largest non-preemptive region allowed for each task τ_i can be set to $Q(d_i)$, as it was in [2]. Preemption points should be placed inside the code of each task so that the worst-case execution time between any two consecutive preemption points do not exceed the computed $Q(d_i)$, for any τ_i. In this case, it is not necessary to store any non-preemptive function for run-time usage, since the desired preemption constraints are enforced off-line. Note that mutual exclusion mechanisms can be easily avoided if preemption points are placed so that each critical section is contained within a non-preemptive region. In this case, no particular locking protocol is needed, further reducing the system overhead.

An exact estimation of the effectiveness of this approach requires the use of appropriate timing analysis tools (see [25] for a good survey on the existing tools). Selecting the best locations in which to place preemption points can significantly reduce the total overhead. We are working on algorithms that can find such optimal placement starting from a set of data provided by an initial timing analysis performed on the task set. Due to the complexity of the analysis, we leave the description of these algorithm as a future work.

8 SIMULATIONS

To evaluate the effectiveness of the proposed algorithm, we performed a series of experiments with various different kinds of randomly generated task sets. We simulated the scheduling of these task sets for a sufficiently large time-interval using EDF and limited-preemption EDF, counting the total number of preemptions for each of the considered algorithms. We expect an overall reduction of context-changes with limited-preemption EDF.

8.1 Experiment setup

We performed a large number of simulations by varying the number of tasks n, the total utilization U and the task parameters: deadline d_i, period p_i and execution time e_i. We considered task sets with $3, 5, 7$ and 10 tasks. For each one of these cases, we let the total utilization vary from 0.1 to 0.9, in steps of size 0.1. Finally, for each considered configuration (n, U), we generated 1000 task sets. For the random generation of the individual task utilizations, we exploited the generation method described by Bini and Buttazzo in [7]. This method allows deriving a uniform distribution of n task utilizations U_i in $[0, 1]$, with total utilization equal to U.

The period p_i of each task is an integer number randomly generated from a uniform distribution in $[10, 1000]$. The execution time e_i is accordingly computed using the generated p_i and U_i: $e_i = \lceil p_i U_i \rceil$. The deadline d_i is a randomly generated integer number from a uniform distribution in $[\max(e_i, p_i/2), 1000]$. We considered the job arrival sequence in which all tasks release an instance at time $t = 0$, and each subsequent instance as soon as legally permitted. Note that different job activations can lead to a higher (or lower) number of preemptions; however, we are more interested in evaluating the *average behavior*, in terms of number of preemptions, of the proposed scheduling algorithm, rather than in finding the exact worst-case number of preemptions of a given sporadic task set.

We counted the number of preemptions ψ^k that occur with EDF and limited-preemption EDF to the k-th generated task set τ^k from the initial time $t = 0$, until time $t = 10^6$. The chosen time interval $[0, 10^6]$ appears sufficiently large to characterize the general behavior of the generated task systems in terms of preemption density. We observed that reducing or increasing the length of the considered interval by one order of magnitude, the computed number of preemptions changes by the same order of magnitude.

The following values have been computed for each considered configuration (n, U):

- The *average number of preemptions* N_{av}, defined as the sum, among all 1000 generated task sets, of the total

number of preemptions experienced by each task set during interval $[0, 10^6]$, divided by the number of generated task sets:

$$N_{av} \doteq \frac{\sum_{k=1}^{1000} \psi^k}{1000}$$

- The *maximum number of preemptions* $N_{\max}$, defined as the maximum among all 1000 generated tasks, of the total number of preemptions experienced by the task set during interval $[0, 10^6]$:

$$N_{\max} \doteq \max_{k=1}^{1000} \psi^k$$

8.2 Evaluation of experiments

In the following histograms, we show the observed average (Figure 2) and maximum (Figure 3) number of preemptions as a function of the task set utilization U, for all considered values of $n \in \{3, 5, 7, 10\}$. The continuous curves represent plain EDF scheduling, while the shaded curves represent our modified algorithm.

From the above figures, it clearly appears that limited-preemption EDF is able to schedule the same task sets that are schedulable with EDF, with a significantly smaller number of preemptions, at all system utilizations. This property becomes more evident considering task sets composed of a large number of tasks. For $n = 10$, our algorithm has an average number of preemptions that is less than 20% that of normal EDF.

A well known property of EDF scheduling is that the number of preemptions in a given interval is bounded by the number of jobs in the same interval [11]. Therefore, increasing the number of tasks while keeping constant the total utilization, the number of jobs increases, as does the number of preemptions. In our simulations, the average number of preemptions with normal EDF seems to be more or less proportional to the number of tasks n. Instead, with limited-preemption EDF, it appears that the average number of preemptions remains more or less constant while varying n.

Both algorithms show a number of context changes roughly proportional to the system utilization. Since the workload is higher at larger utilizations, there are more chances that other tasks arrive while a job is executing. Nevertheless, our modified algorithm shows a very limited number of preemptions even at system utilizations close to 1: with the considered task parameters, the average number of preemptions is never higher than 2×10^4 preemptions every 10^6 time units, for all considered values of n.

Regarding the maximum number of preemptions observed, the difference between the algorithms seems less marked. This means that even if our algorithm exhibits much better behavior on average, it is possible to find task sets for which limited-preemption EDF cannot significantly decrease the number of preemptions. Nevertheless, our algorithm has a maximum number of preemptions that is at least 25% lower than in the EDF case, at almost every system utilization.

In other experiments, we found that increasing the number of task sets generated for each configuration (n, U), the relation between the average and maximum number of preemptions of both algorithms remains more or less unchanged.

The above considerations suggest the use of limited-preemption EDF, whenever it is possible to modify an existing EDF scheduler. A potential complication can be the run-time and memory overhead associated with the management and storage of the array representing the $Q(t)$ function. To mitigate this potential problem, an option is to compute the preemption function only at those points that correspond to a relative deadline among tasks in τ — as mentioned in the previous section — approximating the preemption function $Q(t)$ with the value it assumes at the lowest relative deadline $\geq t$. The memory requirement of this approach is proportional to the number of tasks, as is the complexity of computing the array associated to the preemption function. The experiments we ran with this alternative strategy do not show a significant increase in terms of preemptions: compared to the original algorithm, the average number of preemptions increases by less than 10%, while the maximum number of preemptions remains more or less the same, for every tested configuration. In Figure 4, we show the cases with $n = 3$ and $n = 10$. In the same figure, we plotted as well the curves of the original algorithm described in [2], which uses a static value $q_i = Q(d_i)$ for each task τ_i, independently from the time to deadline at the preemption request. As explained in Section 7.2, this approach is less effective than the other ones, resulting in a further increase in the measured number of preemptions.

In Figure 5, we show the average value of $q_i = Q(d_i)$ for each task τ_i when there are $n = 10$ tasks. Remember that $Q(d_i)$ is a lower bound on the Q-function for τ_i. Therefore, the non-preemptive section length of all presented methods is at least as large as q_i. We show the average among all experiments, and among the most restrictive case in which the total utilization is $U = 0.9$. For comparison, we include the average worst-case execution time C_i for each task. As can be seen, the Q value measured in our experiments is very large; in most cases it is even bigger than the average WCET of the corresponding task. When the task set utilization is large ($U = 0.9$), only two tasks (τ_8 and τ_9) have an average WCET that is slightly greater than the average q_i. Nevertheless, we measured that the average ratio of q_i/C_i is always greater than one, even in the less favorable cases: the average of q_9/C_9 among the experiments with $U = 0.9$ is 1.6. This suggests the possibility of executing even large critical sections with preemptions disabled, as we mentioned in Section 7.3.

Another observation concerns the memory overhead for storing the complete $Q(t)$ function. Such overhead can be reduced by storing only the values at which the preemption-function changes. In our simulations, we found that the points of discontinuity of the $Q(t)$

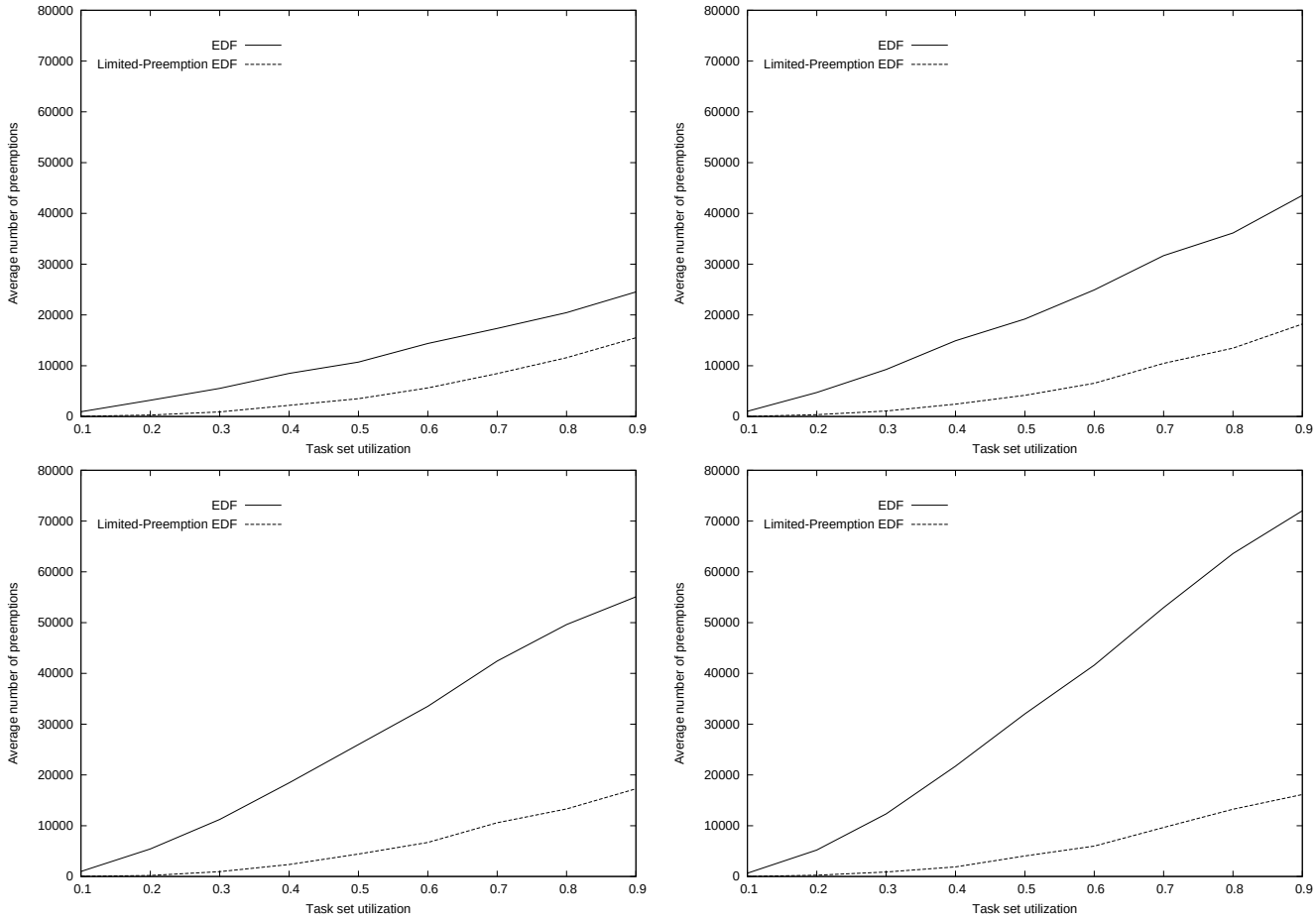

Fig. 2. Average number of preemptions for the experiments with $n = 3$ (top-left), $n = 5$ (top-right), $n = 7$ (bottom-left) and $n = 10$ (bottom-right).

function are very few. In Figure 6, we plot the maximum number of discontinuities of the $Q(t)$ functions, as functions of the task set utilization and of the number of tasks. It appears that these numbers increase for larger U and larger n. Anyway, we never experienced a number of discontinuities greater than 8. Moreover, if we consider the average number of discontinuities among the generated task sets of each configuration (n, U), we find that it is never greater than 3, for all considered configurations. These characteristics allow a very simple and efficient implementation of the limited-preemption EDF algorithm, requiring just a few memory locations to store the required data for the preemption function.

9 Conclusions

From a pragmatic perspective, both preemptive and non-preemptive scheduling have benefits and drawbacks when compared to each other, and neither paradigm is univocally superior to the other. It is desirable to combine features of both paradigms into an integrated scheduling framework that is able to offer benefits that are currently offered separately by the two paradigms.

Towards achieving such an integrated scheduling framework, we have studied the use of *limited preemption* scheduling algorithms. These algorithms do not forbid the use of preemptions entirely; they do, however, discourage the indiscriminate use of preemptions, particularly when not needed to maintain system feasibility. We have designed and analyzed an EDF-based limited-preemption algorithm for scheduling sporadic task systems upon uniprocessor platforms. As expected, this limited-preemption algorithm combines beneficial features from both preemptive and non-preemptive EDF. As with preemptive EDF, limited-preemption EDF has low schedulability overhead and is amenable to efficient feasibility analysis; as with non-preemptive EDF, it offers low run-time overhead and is typically able to offer efficient support for mutual exclusion.

As a future work, we plan to integrate our analysis with the preemption overhead incurred by each task. This overhead influences the non-preemption function, reducing the available slack of each task. Note that with the proposed limited-preemption method, tasks are likely to complete closer to their deadlines, due to the blocking imposed by jobs executing in non-preemptive mode. It is therefore important to select

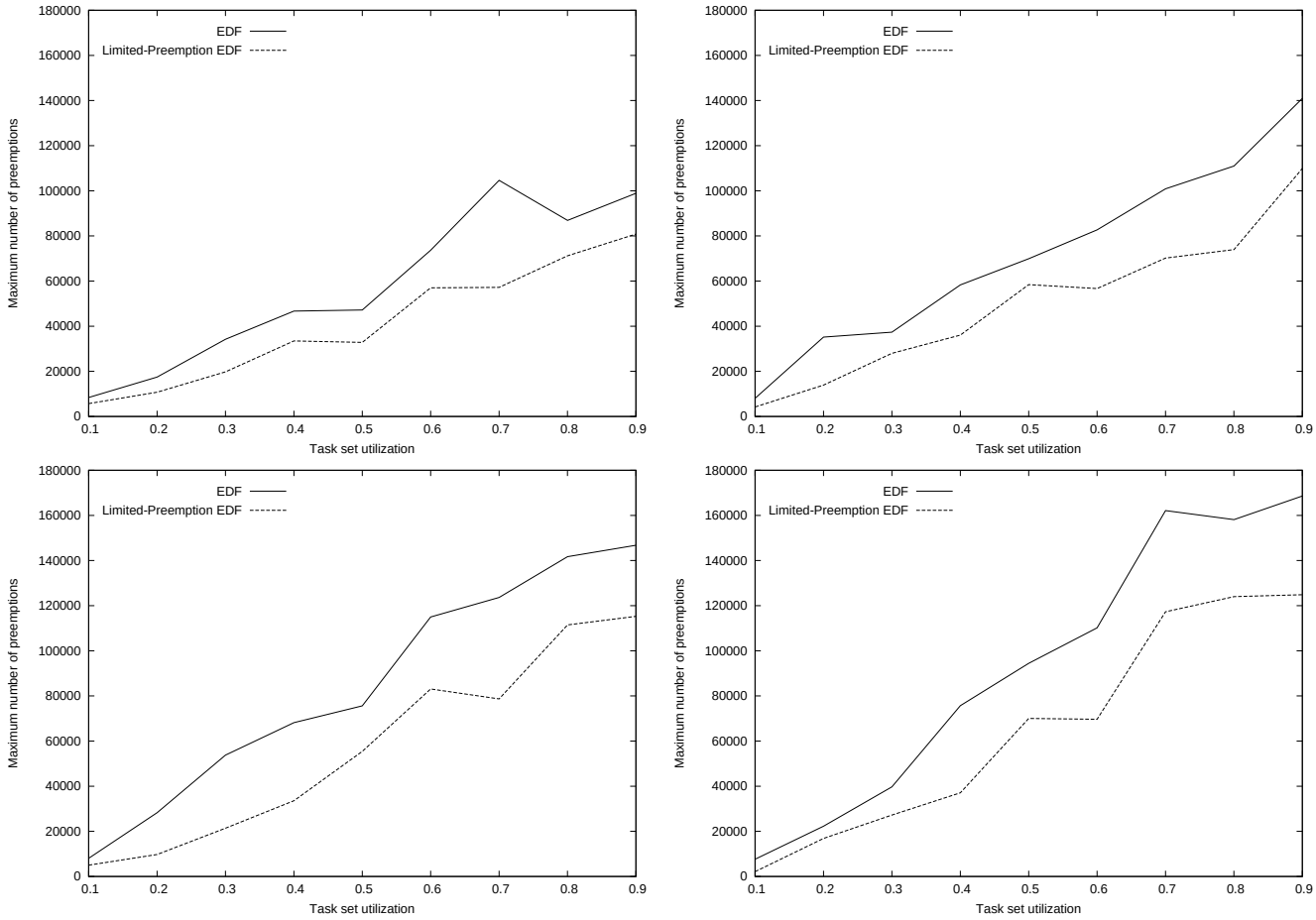

Fig. 3. Maximum number of preemptions for the experiments with $n = 3$ (top-left), $n = 5$ (top-right), $n = 7$ (bottom-left) and $n = 10$ (bottom-right).

reliable WCET bounds to avoid any deadline miss. For now, we assumed the preemption overhead be included in the worst-case execution time e_i of each task τ_i, like in classic preemptive scheduling analysis. Since with limited-preemption scheduling each task will experience a smaller number of context switches, using the same (overestimated) WCET as in the preemptive case will allow for sufficient slack to limit the risk of deadline misses. We are currently working on developing tighter estimation of the preemption overhead, in order to reduce the WCET of each task, increasing the number of task sets that can be scheduled with the proposed method.

REFERENCES

[1] BAKER, T. P. Stack-based scheduling of real-time processes. *Real-Time Systems: The International Journal of Time-Critical Computing 3* (1991).

[2] BARUAH, S. The limited-preemption uniprocessor scheduling of sporadic task systems. In *Proceedings of the EuroMicro Conference on Real-Time Systems* (Palma de Mallorca, Balearic Islands, Spain, July 2005), IEEE Computer Society Press.

[3] BARUAH, S. Resource sharing in EDF-scheduled systems: A closer look. In *Proceedings of the IEEE Real-time Systems Symposium* (Rio de Janeiro, December 2006), IEEE Computer Society Press.

[4] BARUAH, S., AND CHAKRABORTY., S. Schedulability analysis of non-preemptive recurring real-time tasks. In *International Workshop on Parallel and Distributed Real-Time Systems (IPDPS)* (Rhodes, Greece, April 2006).

[5] BARUAH, S., HOWELL, R. R., AND ROSIER, L. E. Feasibility problems for recurring tasks on one processor. *Theoretical Computer Science 118*, 1 (1993), 3–20.

[6] BARUAH, S., MOK, A. K., AND ROSIER, L. E. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th Real-Time Systems Symposium* (Orlando, Florida, 1990), IEEE Computer Society Press.

[7] BINI, E., AND BUTTAZZO, G. C. Biasing effects in schedulability measures. In *ECRTS '04: Proceedings of the 16th Euromicro Conference on Real-Time Systems* (Catania, Italy, July 2004).

[8] BRANDENBURG, B. B., CALANDRINO, J. M., BLOCK, A., LEONTYEV, H., AND ANDERSON, J. H. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *IEEE Real-Time and Embedded Technology and Applications Symposium* (St. Louis, Missouri, USA, 2008), pp. 342–353.

[9] BRIL, R., LUKKIEN, J., AND VERHAEGH, W. 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* (Pisa, Italy, 2007), pp. 269–279.

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

[11] BUTTAZZO, G. C. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park Norwell, MA 02061, USA, 1997.

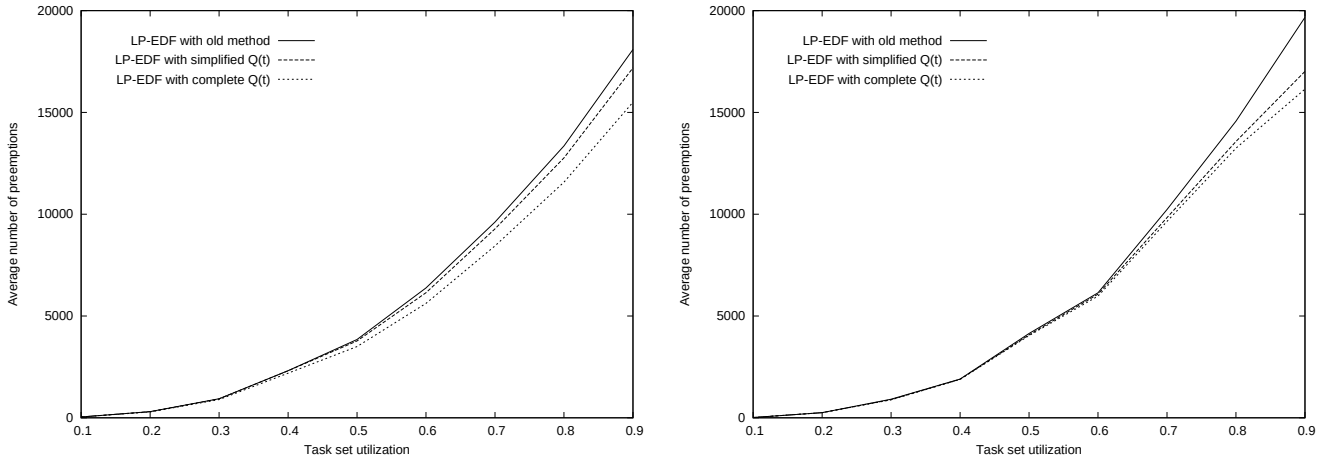

Fig. 4. Average number of preemptions with complete and simplified $Q(t)$ for the experiments with $n = 3$ (left) and $n = 10$ (right).

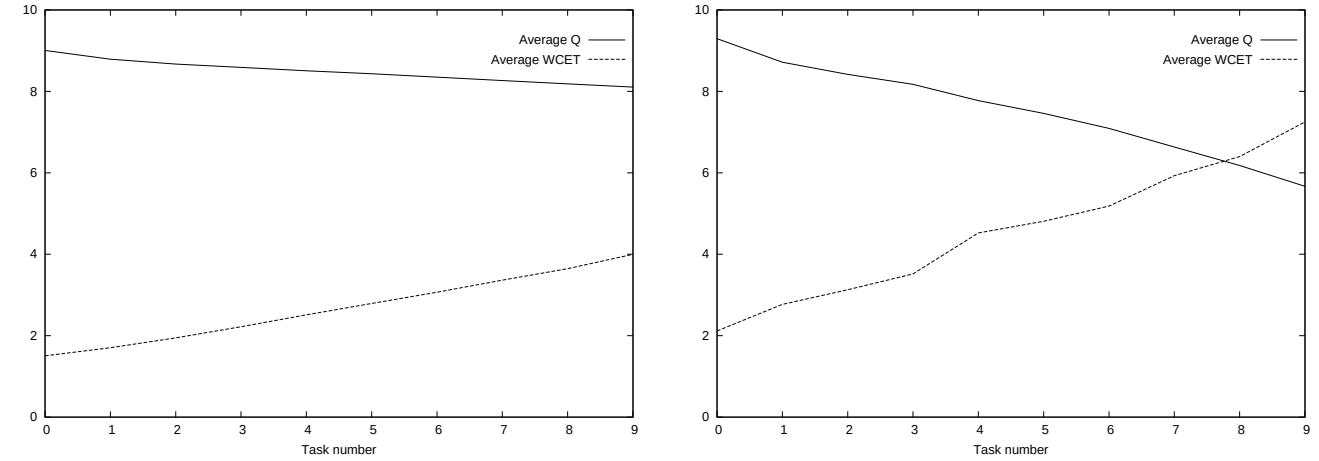

Fig. 5. Average $Q(d_i)$ for each task τ_i when $n = 10$, among all experiments (left) and among experiments with $U = 0.9$ (right).

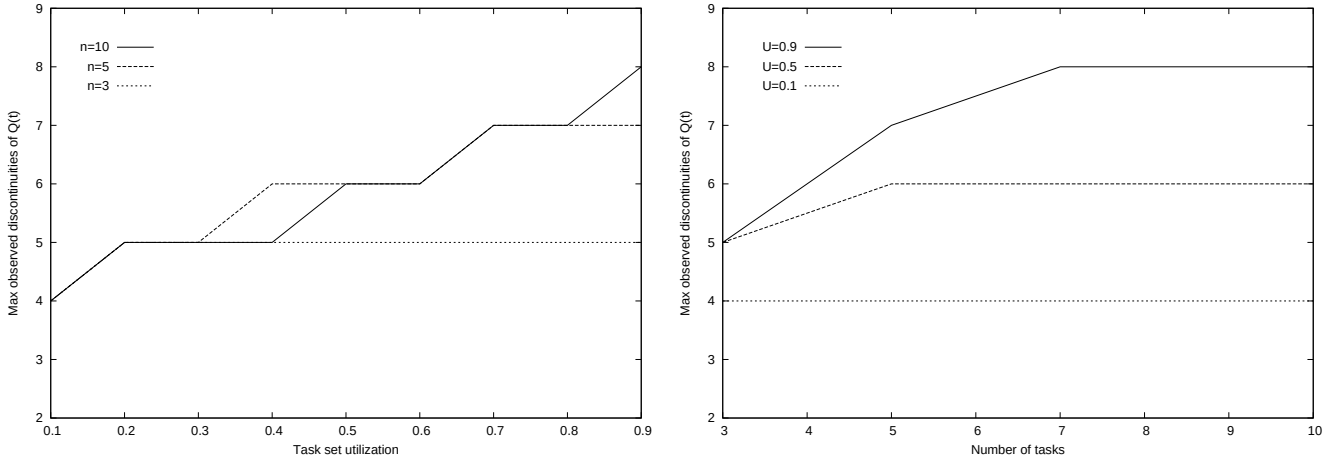

Fig. 6. Maximum number of discontinuities of $Q(t)$ observed in the experiments, as a function of the total utilization (left), and the number of tasks (right).

[12] CHEN, M.-I., AND LIN, K.-J. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Real-Time Systems: The International Journal of Time-Critical Computing 2* (1990).
[13] DERTOUZOS, M. Control robotics : the procedural control of physical processors. In *Proceedings of the IFIP Congress* (1974).
[14] GEORGE, L., RIVIERRE, N., AND SPURI, M. Preemptive and non-preemptive real-time uniprocessor scheduling. Tech. Rep. RR-2966, INRIA: Institut National de Recherche en Informatique et en Automatique, 1996.
[15] GOPALAKRISHNAN, R., AND PARULKAR, G. Bringing real-time scheduling theory and practice closer for multimedia computing. In *Proc. ACM Sigmetrics Conference on Measurement & Modeling of Computer Systems* (May 1996), pp. 1–12.
[16] JEFFAY, K., STANAT, D., AND MARTEL, C. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of the 12th Real-Time Systems Symposium* (San Antonio, Texas, December 1991), IEEE Computer Society Press.
[17] LEUNG, J. Y.-T., AND MERRILL, M. A note on the preemptive scheduling of periodic, real-time tasks. *Information Processing Letters 11* (1980), 115–118.
[18] LIU, C. L., AND LAYLAND, J. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM 20*, 1 (1973), 46–61.
[19] MOK, A. K. *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1983. Available as Technical Report No. MIT/LCS/TR-297.
[20] RAMAMURTHY, S. *A Lock-Free Approach to Object Sharing in Real-Time Systems*. PhD thesis, Department of Computer Science, The University of North Carolina at Chapel Hill, 1997.
[21] REGEHR, J. Scheduling tasks with mixed preemption relations for robustness to timing faults. In *Proceedings of the* 23^{rd} *IEEE Real-Time Systems Symposium* (Cancun (Mexico), December 2002), IEEE Computer Society, pp. 315–326.
[22] SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. P. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers 39*, 9 (1990), 1175–1185.
[23] SIMONSON, J., AND PATEL, J. Use of preferred preemption points in cache-based real-time systems. In *Proc. IEEE International Computer Performance and Dependability Symposium (IPDS)* (April 1995), pp. 316–325.
[24] WANG, Y., AND SAKSENA, M. Scheduling fixed-priority tasks with preemption threshold. In *Proceedings of the International Conference on Real-time Computing Systems and Applications* (1999), IEEE Computer Society.
[25] WILHELM, R., ET AL. The worst-case execution time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS) 7*, 3 (2008), 1–53.
[26] YAO, G., BUTTAZZO, G., AND BERTOGNA, M. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. In *Proceedings of IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)* (Beijing, China, May–June 2009), IEEE Computer Society Press.

Marko Bertogna Marko Bertogna is Assistant Professor at the Scuola Superiore Sant'Anna in Pisa, Italy. He received his Ph.D. in Computer Science from Scuola Superiore Sant'Anna in 2008. He graduated (summa cum laude) in Telecommunications Engineering at the University of Bologna (Italy), in 2002. His research interests include scheduling and schedulability analysis of real-time multiprocessor systems, protocols for the exclusive access to shared resources, hierarchical systems, and reconfigurable devices.

Sanjoy Baruah Sanjoy Baruah is a professor in the Department of Computer Science at the University of North Carolina at Chapel Hill. He received his Ph.D. from the University of Texas at Austin in 1993. His research and teaching interests are in scheduling theory, real-time and safety-critical system design, and resource-allocation and sharing in distributed computing environments.