Limited Preemptive Scheduling for Real-Time Systems. A Survey

Giorgio C. Buttazzo, *Fellow, IEEE*, Marko Bertogna, *Senior Member, IEEE*, and Gang Yao

***Abstract*—The question whether preemptive algorithms are better than nonpreemptive ones for scheduling a set of real-time tasks has been debated for a long time in the research community. In fact, especially under fixed priority systems, each approach has advantages and disadvantages, and no one dominates the other when both predictability and efficiency have to be taken into account in the system design. Recently, limited preemption models have been proposed as a viable alternative between the two extreme cases of fully preemptive and nonpreemptive scheduling. This paper presents a survey of the existing approaches for reducing preemptions and compares them under different metrics, providing both qualitative and quantitative performance evaluations.**

***Index Terms*—Limited-preemptive scheduling, nonpreemptive regions, real-time systems.**

I. Introduction

PREEMPTION is a key factor in real-time scheduling, since it allows the operating system to immediately allocate the processor to incoming tasks requiring urgent service. In fully preemptive systems, the running task can be interrupted at any time by another task with higher priority, and be resumed to continue when all higher priority tasks have completed. In other systems, preemption may be disabled for certain intervals of time during the execution of critical operations (e.g., interrupt service routines, critical sections, etc.). In other situations, preemption can be completely forbidden to avoid unpredictable interference among tasks and achieve a higher degree of predictability (although higher blocking times).

The question whether enabling or disabling preemption during task execution has been investigated by many authors under several points of view and it is not trivial to answer. A general disadvantage of the nonpreemptive discipline is that it introduces additional blocking time in higher priority tasks, so reducing schedulability. On the other hand, there are several advantages to be considered when adopting a nonpreemptive scheduler. In particular, the following issues have to be taken into account when comparing the two approaches.

Manuscript received September 01, 2011; revised November 22, 2011 and February 01, 2012; accepted February 07, 2012. Date of publication March 05, 2012; date of current version December 19, 2012. Paper no. TII-11-501.

G. C. Buttazzo is with the Scuola Superiore Sant'Anna, Pisa 56124, Italy (e-mail. g.buttazzo@sssup.it).

M. Bertogna is with the University of Modena and Reggio Emilia, Modena 41121, Italy (e-mail. marko.bertogna@unimore.it).

G. Yao is with the University of Illinois at Urbana–Champaign, Urbana, IL 61820 USA (e-mail. gangyao@illinois.edu).

Color versions of one or more of the figures in this paper are available online at http.//ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TII.2012.2188805

- In many practical situations, such as I/O scheduling or communication in a shared medium, either preemption is impossible or prohibitively expensive [1].
- Preemption destroys program locality, increasing the runtime overhead due to cache misses and pre-fetch mechanisms. As a consequence, worst-case execution times (WCETs) are more difficult to characterize and predict [2]–[5].
- Mutual exclusion is trivial in nonpreemptive scheduling, which naturally guarantees the exclusive access to shared resources. On the contrary, to avoid unbounded priority inversion [6], preemptive scheduling requires the implementation of specific concurrency control protocols for accessing shared resources [6], [7], which introduce additional overhead and complexity.
- In control applications, the input–output delay and jitter are minimized for all tasks when using a nonpreemptive scheduling discipline, since the interval between start time and finishing time is always equal to the task computation time [8]. This simplifies control techniques for delay compensation at design time.
- Nonpreemptive execution allows using stack sharing techniques [7] to save memory space in small embedded systems with stringent memory constraints [9], [10].

In summary, arbitrary preemptions can introduce a significant runtime overhead and may cause high fluctuations in task execution times, so degrading system predictability. In particular, at least four different types of costs need to be taken into account at each preemption. *Scheduling cost* (for inserting the running task into the ready queue, switch the context, and dispatch the new incoming task); *Pipeline cost* (for flushing the processor pipeline when the task is interrupted and refilling it when the task is resumed); *Cache-related cost* (for reloading the cache lines evicted by the preempting task); *Bus-related cost* (due to the extra bus interference for accessing the RAM because of the additional cache misses caused by preemption). Bui *et al.* [11] showed that on a PowerPC MPC7410 with 2 MByte two-way associative L2 cache the WCET increment due to cache interference can be as large as 33% of the WCET measured in nonpreemptive mode.

The cumulative execution overhead due to the combination of these effects is referred to as *Architecture related cost*. Unfortunately, this cost is characterized by a high variance and depends on the specific point in the task code where preemption takes place [12]–[14].

The total increase of the WCET of a task τ_i is also a function of the total number of preemptions experienced by τ_i, which in turn depends on the task set parameters, on the activation pattern of higher priority tasks, and on the specific scheduling algorithm. Such a circular dependency of WCET and number of preemptions makes the problem not easy to be solved.

The major contribution of this paper is to provide a detailed comparison of the various limited preemptive approaches proposed in the literature, with respect to fully preemptive and non-preemptive schemes. Schedulability tests for each method are reported for completeness, and simulation experiments are carried out to evaluate the impact of the algorithms on the number of preemptions and the overall system schedulability. The results reported here can be used to select the most appropriate scheduling scheme to increase the efficiency of time-critical embedded systems without sacrificing predictability.

The rest of this paper is organized as follows. Section II describes three different approaches proposed in the literature to handle limited preemptive scheduling. Section III describes the task model and the terminology adopted in this paper. Section IV presents the schedulability analysis for the non-preemptive task model. The preemption thresholds model is analyzed in Section V. Section VI details the deferred preemption model, while the method of fixed preemption points is analyzed in Section VII. Considerations regarding the differences between the various models are presented in Section VIII. Section IX reports and discusses some simulation results and Section X states our conclusions.

II. Limited Preemptive Approaches

Often, preemption is considered a prerequisite to meet timing requirement in real-time system design; however, in most cases, a fully preemptive scheduler produces many unnecessary preemptions. To reduce the runtime overhead due to preemptions and still preserve the schedulability of the task set, the following approaches have been proposed in the literature.

- *Preemption Thresholds Scheduling (PTS)*. This approach, proposed by Wang and Saksena [15], allows a task to disable preemption up to a specified priority level, called *preemption threshold*. Thus, each task is assigned a regular priority and a preemption threshold, and the preemption is allowed to take place only when the priority of the arriving task is higher than the threshold of the running task.
- *Deferred Preemptions Scheduling (DPS)*. According to this method, first introduced by Baruah [16] under Earliest Deadline First (EDF), each task τ_i specifies the longest interval q_i that can be executed nonpreemptively. For the sake of clarity, it is worth noting that the terminology is not consistent in the literature, since other authors (e.g., Burns [17] and Bril *et al.* [18]) used the term *Deferred Preemptions* to actually denote *Fixed Preemption Points*. However, we believe the term *Deferred* is more appropriate in this case, because preemption is postponed for a given amount of time, rather than moved to a specific position in the code. Depending on how nonpreemptive regions are implemented, this model can come in two slightly different versions.
  1) *Floating model*. In this model, nonpreemptive regions are defined by the programmer by inserting specific primitives in the task code that disable and enable preemption. However, since the start time of each region is not specified in the model, nonpreemptive regions cannot be identified offline and, for the sake of the analysis, they are considered to be "floating" in the code, with a duration not exceeding q_i.
  2) *Activation-triggered model*. In this model, nonpreemptive regions are triggered by the arrival of a higher priority task and programmed by a timer to last exactly for q_i units of time (unless the task finishes earlier), after which preemption is enabled. Once a timer is set at time t, additional activations arriving before the timeout $(t+q_i)$ do not postpone the preemption any further. Once a preemption takes place, a new high-priority arrival can trigger another nonpreemptive region.
- *Fixed Preemption Points (FPP)*. According to this approach, proposed by Burns [17], a task implicitly executes in nonpreemptive mode and preemption is allowed only at predefined locations inside the task code, called *preemption points*. In this way, a task is divided into a number of nonpreemptive chunks (also called subjobs). If a higher priority task arrives between two preemption points of the running task, preemption is postponed until the next preemption point. This approach is also referred to as *Cooperative scheduling*, because tasks cooperate to offer suitable preemption points to improve schedulability.

III. Terminology and Notation

Let us consider a set of n periodic or sporadic real-time tasks that need to be scheduled on a single processor. Each task τ_i is characterized by a WCET C_i, a relative deadline D_i, and a period (or minimum interarrival time) T_i. A constrained deadline model is adopted, so D_i is assumed to be less than or equal to T_i. Each task is assigned a fixed priority P_i, used to select the running task among those tasks ready to execute (a higher value of P_i corresponds to a higher priority). Notice that task activation times are not known *a priori* and the actual execution time of a task can be less than or equal to its worst-case value C_i. Tasks are indexed by decreasing priority, i.e., $\forall i | 1 \leq i < n.P_i > P_{i+1}$. Additional terminology will be introduced below for each specific method.

A. Integer Time Model

In real-time operating systems, time instants and interval durations are measured by counting the number of clock cycles generated by a real-time clock, hence all timing values have a resolution equal to one clock cycle. Therefore, to use analytical results in a real embedded system, all timing parameters are assumed to be non-negative integer values. To comply with such a convention, all cited results derived under the domain of real numbers have been adapted to the integer time model.

B. Critical Instant

The maximum response time of each task is derived under the worst-case arrival pattern that leads to the largest interference on the considered task. Such a particular scenario is often referred to as the *critical instant*. For fully preemptive fixed priority systems, Liu and Layland [19] proved that the critical instant for a task occurs when it arrives synchronously with all higher priority tasks, and all task instances are released as soon as possible, i.e., in a strictly periodic fashion.

TABLE I
PARAMETERS OF A SAMPLE TASK SET

	C_i	T_i	D_i
τ_1	1	6	4
τ_2	3	10	9
τ_3	6	18	12

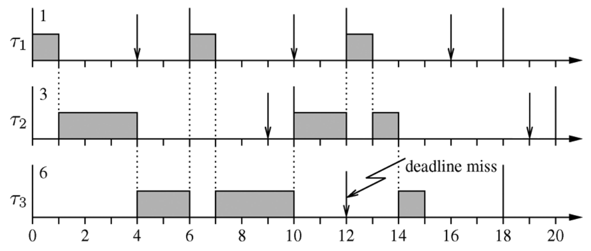


Fig. 1. Schedule produced by Deadline Monotonic (in fully preemptive mode) on the task set of Table I.

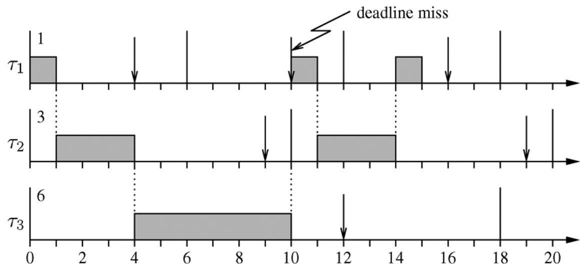


Fig. 2. Schedule produced by nonpreemptive deadline monotonic on the task set of Table I.

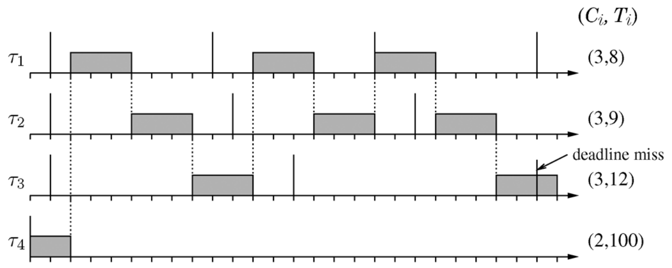


Fig. 3. An example of self-pushing phenomenon occurring on task τ_3.

In the presence of nonpreemptive regions, however, the additional blocking from lower priority tasks has to be taken into account, hence, the critical instant for a task τ_i occurs when it is released synchronously and periodically with all higher priority tasks, while the lower priority task that is responsible of the largest blocking time of τ_i is released one unit of time before τ_i [20]. However, the largest response time of a task is not necessarily due to the first job after a critical instant, but might be due to later jobs, as explained later on.

All schedulability tests hereafter presented have been derived by computing the worst-case response time of a task under the above described notion of critical instant.

C. *Motivating Example*

To better appreciate the importance of limited preemptive scheduling and to better understand the difference among the limited preemptive approaches presented in this survey, Table I reports a sample task set that will be used as a common example throughout this paper, because it results to be unschedulable by Deadline Monotonic [21], both in fully preemptive and in fully nonpreemptive mode, but it can be schedulable by all limited preemptive approaches.

Fig. 1 illustrates the schedule produced by Deadline Monotonic in fully preemptive mode. As clear from the figure, the task set is not feasible, since task τ_3 misses its deadline.

IV. NONPREEMPTIVE SCHEDULING (NPS)

The most effective way to reduce preemption cost is to disable preemptions completely. In this condition, however, each task τ_i can experience a blocking time B_i equal to the longest computation time among the tasks with lower priority. That is

$$B_i = \max_{j.P_j<P_i} \{C_j - 1\} \tag{1}$$

where the maximum of an empty set is assumed to be zero. Notice that one unit of time is subtracted from the computation time of the blocking task to consider that, to block τ_i, it must start at least one unit before the critical instant. Such a blocking term introduces an additional delay before task execution, which could jeopardize schedulability. High-priority tasks are those that are most affected by such a blocking delay, since the maximum in (1) is computed over a larger set of tasks. Fig. 2 illustrates the schedule generated by Deadline Monotonic on the task set of Table I when preemptions are disabled. With respect to the schedule shown in Fig. 1, notice that τ_3 is now able to complete before its deadline, but the task set is still not schedulable, since now τ_1 misses its deadline.

Unfortunately, under nonpreemptive scheduling, the least upper bounds of both Rate Monotonic (RM) [19] and EDF [19] drop to zero! This means that there exist task sets with arbitrary low utilization that cannot be scheduled by RM and EDF when preemptions are disabled.

A. *Feasibility Analysis*

The feasibility analysis of nonpreemptive task sets is more complex than under fully preemptive scheduling. Davis *et al.* [22] showed that in nonpreemptive scheduling the largest response time of a task does not necessarily occur in the first job, after the critical instant. An example of such a situation is illustrated in Fig. 3, where the worst-case response time of τ_3 occurs in its second instance. Such a scheduling anomaly, identified as *self-pushing phenomenon*, occurs because the high-priority jobs activated during the nonpreemptive execution of τ_i's first instance are pushed ahead to successive jobs, which then may experience a higher interference.

The presence of the self-pushing phenomenon in nonpreemptive scheduling implies that the response time analysis for a task τ_i cannot be limited to its first job, activated at the critical instant, as done in preemptive scheduling, but it must be performed for multiple jobs, until the processor finishes executing tasks with priority higher than or equal to P_i. Hence, the response time of a task τ_i needs to be computed within the longest Level-i Active Period, defined as follows [20].

Definition 1: The *Level-i pending workload* $W_i^p(t)$ at time t is the amount of processing that still needs to be performed at time t due to jobs with priority higher than or equal to P_i released strictly before t.

Definition 2: A *Level-i Active Period* L_i is an interval $[a, b)$ such that the Level-i pending workload $W_i^p(t)$ is positive for all $t \in (a, b)$ and null in a and b.

The longest Level-i Active Period can be computed by the following recurrent relation:

$$\begin{cases} L_i^{(0)} = B_i + C_i \\ L_i^{(s)} = B_i + \sum_{h.P_h \geq P_i} \left\lceil \frac{L_i^{(s-1)}}{T_h} \right\rceil C_h. \end{cases} \tag{2}$$

In particular, L_i is the smallest value for which $L_i^{(s)} = L_i^{(s-1)}$. This means that the response time of task τ_i must be computed for all jobs $\tau_{i,k}$, with $k \in [1, K_i]$, where

$$K_i = \left\lceil \frac{L_i}{T_i} \right\rceil. \tag{3}$$

For a generic job $\tau_{i,k}$, the start time $s_{i,k}$ can then be computed considering the blocking time B_i, the computation time of the preceding $(k-1)$ jobs and the interference of the tasks with priority higher than P_i. Hence, $s_{i,k}$ can be computed with the following recurrent relation:

$$\begin{cases} s_{i,k}^{(0)} = B_i + \sum\limits_{h.P_h > P_i} C_h \\ s_{i,k}^{(\ell)} = B_i + (k-1)C_i + \sum\limits_{h.P_h > P_i} \left(\left\lfloor \frac{s_{i,k}^{(\ell-1)}}{T_h} \right\rfloor + 1 \right) C_h. \end{cases} \tag{4}$$

Note that the original result derived in [20] adopted two different expressions, one for the nth task, that does not experience any blocking, and one for the remaining tasks. Instead, using an integer time model and computing the blocking term with (1), it is possible to simplify the analysis, using a homogeneous formulation for all tasks.

Since, once started, the task cannot be preempted, the finishing time $f_{i,k}$ can be computed as

$$f_{i,k} = s_{i,k} + C_i. \tag{5}$$

Hence, the response time of task τ_i is given by

$$R_i = \max_{k \in [1, K_i]} \{f_{i,k} - (k-1)T_i\}. \tag{6}$$

Once the response time of each task is computed, the task set is feasible if and only if

$$\forall i = 1, \ldots, n \quad R_i \leq D_i. \tag{7}$$

Yao *et al.* [23] showed that the analysis of nonpreemptive tasks can be reduced to analyzing a single job, under specific (but not too restrictive) conditions. The following theorem, originally stated for the fixed preemption model, is presented here for the nonpreemptive scheduling model, which is a special case of the fixed preemption model.

Theorem 1 (From [23]): The worst-case response time of a nonpreemptive task occurs in the first job after its critical instant if the following two conditions are both satisfied:

1) the task set is feasible under preemptive scheduling;
2) relative deadlines are less than or equal to periods.

Under these conditions, the longest relative start time S_i of task τ_i is equal to $s_{i,1}$ and can be computed from (4) for $k = 1$

$$\begin{cases} S_i^{(0)} = B_i + \sum\limits_{h.P_h > P_i} C_h \\ S_i^{(\ell)} = B_i + \sum\limits_{h.P_h > P_i} \left(\left\lfloor \frac{S_i^{(\ell-1)}}{T_h} \right\rfloor + 1 \right) C_h. \end{cases} \tag{8}$$

Hence, the response time R_i is simply

$$R_i = S_i + C_i. \tag{9}$$

V. Preemption Thresholds Scheduling (PTS)

According to this model, proposed by Wang and Saksena [15], each task τ_i is assigned a nominal priority P_i (used to enqueue the task into the ready queue and to preempt) and a *preemption threshold* $\theta_i \geq P_i$ (used for task execution). Then, τ_i can be preempted by τ_h only if $P_h > \theta_i$. At the activation time $r_{i,k}$, the priority of τ_i is set to its nominal value P_i, so it can preempt all the tasks τ_j with threshold $\theta_j < P_i$. The nominal priority is maintained as long as the task is kept in the ready queue. During this interval, τ_i can be delayed by all tasks τ_h with priority $P_h > P_i$ and by at most one lower priority task τ_l with threshold $\theta_l \geq P_i$. When all such tasks complete (at time $s_{i,k}$), τ_i is dispatched for execution and its priority is raised at its threshold level θ_i until the task terminates (at time $f_{i,k}$). During this interval, τ_i can be preempted by all tasks τ_h with priority $P_h > \theta_i$. Notice that, when τ_i is preempted, its priority is kept to its threshold level.

Preemption threshold can be considered as a tradeoff between fully preemptive and fully nonpreemptive scheduling. Indeed, if each threshold priority is set equal to the task nominal priority, the scheduler behaves like a fully preemptive scheduler; whereas, if all thresholds are set to the maximum priority, the scheduler runs in a nonpreemptive fashion. Wang and Saksena also showed that, by appropriately setting the thresholds, the system can improve the schedulability compared with fully preemptive and fully nonpreemptive scheduling.

For example, if priorities are assigned as $P_1 = 3$, $P_2 = 2$, and $P_3 = 1$, and thresholds as $\theta_1 = 3$, $\theta_2 = 3$, and $\theta_3 = 2$, the task set of Table I results to be schedulable, and the schedule produced in the synchronous periodic arrival pattern is illustrated in Fig. 4.

Notice that, at $t = 6$, τ_1 can preempt τ_3 since $P_1 > \theta_3$. However, at $t = 10$, τ_2 cannot preempt τ_3, being $P_2 = \theta_3$. Similarly, at $t = 12$, τ_1 cannot preempt τ_2, being $P_1 = \theta_2$.

A. Feasibility Analysis

Under fixed priorities, the feasibility analysis of a task set with preemption thresholds can be performed by the test derived

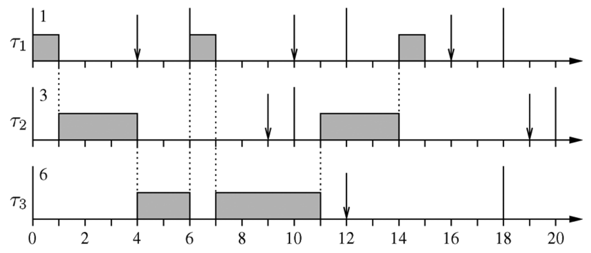


Fig. 4. Schedule produced by Deadline Monotonic on the task set in Table I with priorities $P_1 = 3$, $P_2 = 2$, and $P_3 = 1$, and thresholds $\theta_1 = 3$, $\theta_2 = 3$, and $\theta_3 = 2$.

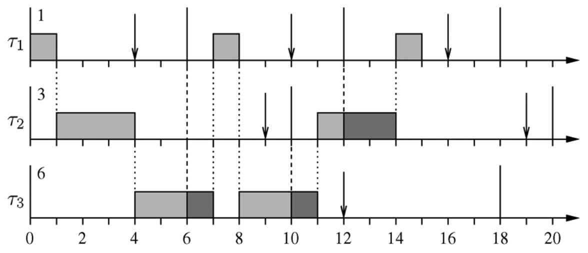


Fig. 5. Schedule produced by deadline monotonic with deferred preemptions for the task set reported in Table I, with $q_2 = 2$ and $q_3 = 1$.

by Keskin *et al.* [24].[1] First of all, a task τ_i can be blocked only by lower priority tasks that cannot be preempted by it, that is, by tasks τ_j with $P_j < P_i$ and $\theta_j \geq P_i$. Hence, a task τ_i can experience a blocking time equal to the longest computation time among the tasks with priority lower than P_i and threshold higher than or equal to P_i. That is

$$B_i = \max_j \{C_j - 1 | P_j < P_i \leq \theta_j\} \tag{10}$$

where the maximum of an empty set is assumed to be zero. Then, the response time R_i of task τ_i is computed by considering the blocking time B_i, the interference before its start time (due to the tasks with priority higher than P_i), and the interference after its start time (due to tasks with priority higher than θ_i). The analysis must be carried out within the longest Level-i active period L_i defined in (2). This means that the response time of task τ_i must be computed for all the jobs $\tau_{i,k}$ with $k \in [1, K_i]$, where K_i is defined in (3).

For a generic job $\tau_{i,k}$, the start time $s_{i,k}$ can be computed considering the blocking time B_i, the computation time of the preceding $(k-1)$ jobs, and the interference of the tasks with priority higher than P_i. Hence, $s_{i,k}$ can be computed using (4). The finishing time $f_{i,k}$ can be computed by summing to the start time $s_{i,k}$ the computation time of job $\tau_{i,k}$, and the interference of the tasks that can preempt $\tau_{i,k}$ (those with priority higher than θ_i). That is

$$\begin{cases} f_{i,k}^{(0)} = s_{i,k} + C_i \\ f_{i,k}^{(\ell)} = s_{i,k} + C_i + \sum\limits_{h.P_h > \theta_i} \left(\left\lceil \frac{f_{i,k}^{(\ell-1)}}{T_h} \right\rceil - \left(\left\lfloor \frac{s_{i,k}}{T_h} \right\rfloor + 1 \right) \right) C_h. \end{cases} \tag{11}$$

Again, the integer time model adopted in this paper, along with the convention on the blocking term given by (10), allow simplifying the analysis with respect to [24], without needing to use two different expressions for the cases with and without blocking.

The response time of task τ_i can then be computed using (6), and the task set is feasible if and only if Condition (7) is satisfied.

The feasibility analysis under preemption thresholds can also be simplified under the conditions of Theorem 1. In this case, we have that the worst-case start time is computed using (8), and the worst-case response time of task τ_i can be computed as

$$\begin{cases} R_i^{(0)} = S_i + C_i + \sum\limits_{h.P_h > \theta_i} C_h \\ R_i^{(\ell)} = S_i + C_i + \sum\limits_{h.P_h > \theta_i} \left(\left\lceil \frac{R_i^{(\ell-1)}}{T_h} \right\rceil - \left(\left\lfloor \frac{S_i}{T_h} \right\rfloor + 1 \right) \right) C_h. \end{cases} \tag{12}$$

[1]The original analysis by Wang and Saksena [15] has been corrected by Regehr [25], which in its turn has been improved by Keskin *et al.* [24].

B. Selecting Preemption Thresholds

The example illustrated in Fig. 4 shows that a task set unfeasible under both preemptive and nonpreemptive scheduling can be feasible under preemption thresholds, for a suitable setting of threshold levels.

Given a task set that is feasible under preemptive scheduling, an interesting problem is to determine the thresholds that limit preemption as much as possible, without jeopardizing the schedulability of the task set. Saksena and Wang [26] proposed an algorithm to increase the threshold of each task up to the level after which the schedule would become infeasible. The algorithm considers one task at the time, starting from the highest priority task.

VI. Deferred Preemptions Scheduling (DPS)

According to this method, each task τ_i defines a maximum interval of time q_i in which it can execute nonpreemptively. Depending on the specific implementation, the nonpreemptive interval can start after the invocation of a system call inserted at the beginning of a nonpreemptive region (floating model), or can be triggered by the arrival of a higher priority task (activation-triggered model).

Under the floating model, preemption is resumed by another system call, inserted at the end of the region (at most q_i units long); whereas, under the activation-triggered model, preemption is enabled by a timer interrupt after exactly q_i units (unless the task completes earlier). For example, considering the same task set of Table I, assigning $q_2 = 2$ and $q_3 = 1$, the schedule produced by Deadline Monotonic with deferred preemptions under the activation-triggered model is feasible, as illustrated in Fig. 5. Dark regions represent nonpreemptive intervals triggered by the arrival of higher priority tasks.

A. Feasibility Analysis

In the presence of nonpreemptive intervals, a task can be blocked when, at its arrival, a lower priority task is running in nonpreemptive mode. Since each task can be blocked at most once by a single lower priority task, B_i is equal to the longest nonpreemptive interval belonging to tasks with lower priority. In particular, the blocking factor can be computed as

$$B_i = \max_{j.P_j < P_i} \{q_j - 1\}. \tag{13}$$

Note that, under the floating model, one unit of time must be subtracted from q_j to allow the nonpreemptive region to start before τ_i. Under the activation-triggered model, however, there is no need to subtract one unit of time from q_j, since the nonpreemptive interval is programmed to be exactly q_j from the arrival time of a higher priority task.

In both the floating and activation-triggered cases, the start times of nonpreemptive intervals are assumed to be unknown *a priori*. Therefore, nonpreemptive regions cannot be identified offline and, for the sake of the analysis, they are considered to occur in the worst possible time (in the sense of schedulability). Then, schedulability can be checked through the classic response time analysis

$$R_i = B_i + \sum_{h.P_h \geq P_i} \left\lceil \frac{R_i}{T_h} \right\rceil C_h. \tag{14}$$

Note that, under the floating model, the analysis does not need to be carried out within the longest Level-i active period. In fact, the worst-case interference on τ_i occurs in the first instance assuming that τ_i could be preempted one time-unit before its completion.

On the other hand, the analysis is more pessimistic under the activation-triggered model, where nonpreemptive intervals are exactly equal to q_i units and can last until the end of the task. In this case, the analysis does not take advantage of the fact that τ_i cannot be preempted when higher periodic tasks arrive q_i units (or less) before its completion. The advantage of such a pessimism, however, is that the analysis is much simpler and can be limited to the first job of each task. Under these assumptions, a task set is feasible with deferred preemptions only if the task set is feasible preemptively. The analysis of periodic tasks with floating nonpreemptive regions has also been developed under EDF [27], [28].

B. Longest Nonpreemptive Interval

When using the deferred preemption method, an interesting problem is to find the longest nonpreemptive interval Q_i for each task τ_i that can still preserve the task set schedulability. More precisely, the problem can be stated as follows.

> Given a set of n periodic tasks that is feasible under preemptive scheduling, find the longest nonpreemptive interval of length Q_i for each task τ_i, so that τ_i can continue to execute for Q_i units of time in nonpreemptive mode, without violating the schedulability of the original task set.

This problem has been solved under EDF by Bertogna and Baruah [27], and under fixed priorities by Yao *et al.* [29]. The solution is based on the concept of *blocking tolerance* β_i, for a task τ_i, defined as follows.

Definition 3: The blocking tolerance β_i of a task τ_i is the maximum amount of blocking τ_i can tolerate without missing any of its deadlines.

When deadlines are equal to periods, a simple way to compute a lower bound on the blocking tolerance is from the Liu and Layland test [19], which, in the presence of blocking factors, becomes

$$\forall i = 1, \ldots, n \qquad \sum_{h.P_h \geq P_i} \frac{C_h}{T_h} + \frac{B_i}{T_i} \leq U_{lub}(i)$$

where $U_{lub}(i) = i(2^{1/i} - 1)$. Isolating the blocking factor, the test can also be rewritten as

$$B_i \leq T_i \left[U_{lub}(i) - \sum_{h.P_h \geq P_i} \frac{C_h}{T_h} \right].$$

Hence, considering integer computations, we have

$$\beta_i = \left\lfloor T_i \left(U_{lub}(i) - \sum_{h.P_h \geq P_i} \frac{C_h}{T_h} \right) \right\rfloor. \tag{15}$$

When deadlines are less than or equal to periods, an exact bound for β_i can instead be achieved by using the schedulability test presented in [30], so that a task set is schedulable with deferred preemptions if and only if for each task τ_i

$$\exists t \in \mathcal{TS}(\tau_i). \quad B_i + \sum_{h.P_h \geq P_i} \left\lceil \frac{t}{T_h} \right\rceil C_h \leq t \tag{16}$$

where

$$\mathcal{TS}(\tau_i) \overset{\text{def}}{=} \mathcal{P}_{i-1}(D_i) \tag{17}$$

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

$$\begin{cases} \mathcal{P}_0(t) = \{t\} \\ \mathcal{P}_i(t) = \mathcal{P}_{i-1}\left(\left\lfloor \frac{t}{T_i} \right\rfloor T_i \right) \cup \mathcal{P}_{i-1}(t). \end{cases} \tag{18}$$

This leads to the following result:[29]

$$B_i \leq \max_{t \in \mathcal{TS}(\tau_i)} \left\{ t - \sum_{h.P_h \geq P_i} \left\lceil \frac{t}{T_h} \right\rceil C_h \right\}.$$

$$\beta_i = \max_{t \in \mathcal{TS}(\tau_i)} \left\{ t - \sum_{h.P_h \geq P_i} \left\lceil \frac{t}{T_h} \right\rceil C_h \right\}. \tag{19}$$

Given the blocking tolerance, the feasibility test can also be expressed as follows:

$$\forall i = 1, \ldots, n \quad B_i \leq \beta_i$$

and, by (13), we can write

$$\forall i = 1, \ldots, n \quad \max_{j.P_j < P_i} \{q_j - 1\} \leq \beta_i.$$

This implies that, to achieve feasibility, we must have

$$\forall i = 1, \ldots, n \quad q_i \leq \min_{k.P_k > P_i} \{\beta_k + 1\}.$$

Hence, the longest nonpreemptive interval Q_i that preserves feasibility for each task τ_i is

$$Q_i = \min_{k.P_k > P_i} \{\beta_k + 1\}. \tag{20}$$

The Q_i terms can also be computed more efficiently, starting from the highest priority task (τ_1) and proceeding with decreasing priority order, according to the following theorem.

Theorem 2 (From [29]): The longest nonpreemptive interval Q_i of task τ_i that preserves feasibility can be computed as

$$Q_i = \min\{Q_{i-1}, \beta_{i-1} + 1\} \tag{21}$$

where $Q_1 = \infty$ and $\beta_1 = D_1 - C_1$.

Note that, in order to apply Theorem 2, Q_i is not constrained to be less than or equal to C_i, but a value of Q_i greater than C_i means that τ_i can be fully executed in nonpreemptive mode.

VII. Fixed Preemption Points (FPP)

According to this model, each task τ_i is split into m_i nonpreemptive chunks (subjobs), obtained by inserting $m_i - 1$ preemption points in the code. Thus, preemptions can only occur at the subjobs boundaries. All the jobs generated by one task have the same subjob division. The k^{th} subjob has a WCET $q_{i,k}$, hence $C_i = \sum_{k=1}^{m_i} q_{i,k}$.

Among all the parameters describing the subjobs of a task, two values are of particular importance for achieving a tight schedulability result

$$\begin{cases} q_i^{\max} = \max_{k\in[1,m_i]}\{q_{i,k}\} \\ q_i^{\text{last}} = q_{i,m_i} \end{cases} \tag{22}$$

In fact, the feasibility of a high-priority task τ_k is affected by the size $q_j^{\max}$ of the longest subjob of each task τ_j with priority $P_j < P_k$. Moreover, the length q_i^{last} of the final subjob of τ_i 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. Therefore, in this model, each task will be characterized by the following 5-tuple:

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

For example, consider the same task set of Table I, and suppose that τ_2 is split into two subjobs of 2 and 1 unit, and τ_3 is split into two subjobs of 4 and 2 units. The schedule produced by Deadline Monotonic with such a splitting is feasible and it is illustrated in Fig. 6.

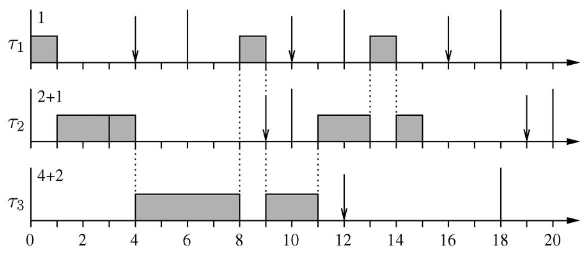


Fig. 6. Schedule produced by Deadline Monotonic for the task set reported in Table I, when τ_2 is split into two subjobs of 2 and 1 unit, and τ_3 is split into two subjobs of 4 and 2 units, respectively.

A. Feasibility Analysis

Feasibility analysis for tasks with fixed preemption points can be carried out in a very similar way as the nonpreemptive case, with the following differences.

- The blocking factor B_i to be considered for each task τ_i is equal to the length of longest subjob (instead of the longest task) among those with lower priority

$$B_i = \max_{j.P_j < P_i} \left\{q_j^{\max} - 1\right\}. \tag{23}$$

- The last nonpreemptive chunk of τ_i is equal to q_i^{last} (instead of C_i).

The response time analysis for a task τ_i has to consider all the jobs within the longest Level-i Active Period, defined in (2). This means that the response time of τ_i must be computed for all jobs $\tau_{i,k}$ with $k \in [1, K_i]$, where K_i is defined in (3).

For a generic job $\tau_{i,k}$, the start time $s_{i,k}$ of the last subjob can be computed considering the blocking time B_i, the computation time of the preceding $(k-1)$ jobs, the subjobs of $\tau_{i,k}$ preceding the last one $(C_i - q_i^{\text{last}})$, and the interference of the tasks with priority higher than P_i. Hence, $s_{i,k}$ can be computed with the following recurrent relation:

$$\begin{cases} s_{i,k}^{(0)} = B_i + C_i - q_i^{\text{last}} + \sum\limits_{h.P_h > P_i} C_h \\ s_{i,k}^{(\ell)} = B_i + kC_i - q_i^{\text{last}} + \sum\limits_{h.P_h > P_i} \left(\left\lfloor \frac{s_{i,k}^{(\ell-1)}}{T_h} \right\rfloor + 1 \right) C_h. \end{cases} \tag{24}$$

Also, in this case, the original result reported by Bril *et al.* [20] adopted a more complex expression, separating the lowest priority task from the higher priority ones. The expression presented here has been simplified thanks to the integer time model, provided the blocking term is computed using (23).

Since, once started, the last subjob cannot be preempted, the finishing time $f_{i,k}$ can be computed as

$$f_{i,k} = s_{i,k} + q_i^{\text{last}}. \tag{25}$$

The response time of task τ_i can then be computed using (6), and the task set is feasible if and only if Condition (7) is satisfied.

B. Longest Nonpreemptive Interval

As done in Section VI-B under deferred preemptions, it is interesting to compute, also under task splitting, the longest nonpreemptive interval Q_i for each task τ_i that can guarantee the schedulability. It is worth observing that splitting tasks into subjobs allows achieving a larger Q_i, because a task τ_i cannot be preempted during the execution of the last q_i^{last} units of time. As shown by Bertogna *et al.* [31], there are cases in which Q_i

can be computed even when the task set is not preemptively feasible, because the last nonpreemptive region allows reducing the interference from higher priority tasks.

Defining $\beta_{i,k}$ as the blocking tolerance of the kth job of τ_i after a critical instant, the schedulability of such a job can be checked using the following condition [31]:

$$\exists t \in \Pi_{i,k}.B_i \leq t - kC_i + q_i^{\text{last}} - \sum_{h.P_h > P_i} \left(\left\lfloor \frac{t}{T_h} \right\rfloor + 1 \right) C_h \tag{26}$$

where

$$\Pi_{i,k} \overset{\text{def}}{=} \left((k-1)T_i, (k-1)T_i + D_i - q_i^{\text{last}}\right] \cap \{hT_{j-1}, \forall h \in \mathbb{N}, j \leq i\} \cup \left\{(k-1)T_i + D_i - q_i^{\text{last}}\right\}.$$

Hence, the blocking tolerance $\beta_{i,k}$ becomes

$$\beta_{i,k} = \max_{t \in \Pi_{i,k}} \left\{ t - kC_i + q_i^{\text{last}} - \sum_{h.P_h > P_i} \left(\left\lfloor \frac{t}{T_h} \right\rfloor + 1 \right) C_h \right\}. \tag{27}$$

The blocking tolerance of task τ_i can be computed as the minimum blocking tolerance among the first K_i jobs of τ_i in the longest Level-i Active Period

$$\beta_i = \min_{k \in [1, K_i]} \beta_{i,k} \tag{28}$$

where K_i is defined in (3).

From (27), it is easy to see that the blocking tolerances $\beta_{i,k}$ do not depend on B_i, which can be set to $\beta_{i,1}$ without affecting the analysis. The longest nonpreemptive interval Q_i that guarantees the feasibility for each task τ_i can then be computed by Theorem 2, using the blocking tolerances given by (28).

As previously mentioned, the maximum length of the nonpreemptive chunk under fixed preemption points is larger than in the case of deferred preemptions. It is worth pointing out that the value of Q_i for task τ_i only depends on the β_k of the higher priority tasks, as expressed in (20), and the blocking tolerance β_i is a function of q_i^{last}, as clear from (28). Note that when tasks are assumed to be preemptively feasible, the analysis can be limited to the first job of each task. In this case, the blocking tolerance of task τ_i is $\beta_i = \beta_{i,1}$.

VIII. Assessment of the Approaches

The limited preemption methods presented in this paper can be compared under several aspects, such as. implementation complexity, predictability in estimating the preemption cost, effectiveness in improving schedulability, and in reducing the number of preemptions.

A. Implementation Issues

The preemption threshold mechanism can be implemented by raising the execution priority of the task, as soon as it starts running. The mechanism can be easily implemented at the application level by calling, at the beginning of the task, a system call that increases the priority of the task at its threshold level. The mechanism can also be fully implemented at the operating system level, without modifying the application tasks. To do that, the kernel has to increase the priority of a task at the level of its threshold when the task is scheduled for the first time. In this way, at its first activation, a task is inserted in the ready queue using its nominal priority. Then, when the task is scheduled for execution, its priority becomes equal to its threshold, until completion. Note that, if a task is preempted, its priority remains at its threshold level.

Note that preemption threshold scheduling is already used in the ThreadX real-time operating system by Express Logic Inc., and in the Erika Enterprise real-time kernel by Evidence, and it represents an example of a great success of transferring research results to industrial applications.

In deferred preemption (floating model), nonpreemptive regions can be implemented by proper kernel primitives that disable and enable preemption at the beginning and at the end of the region, respectively. As an alternative, preemption can be disabled by increasing the priority of the task at its maximum value, and can be enabled by restoring the nominal task priority. In the activation-triggered mode, nonpreemptive regions can be realized by setting a timer to enforce the maximum interval in which preemption is disabled. Initially, all tasks start executing in nonpreemptive mode. When τ_i is running and a task with priority higher than P_i is activated, a timer is set by the kernel (inside the activation primitive) to interrupt τ_i after q_i units of time. Until then, τ_i continues executing in nonpreemptive mode. The interrupt handler associated to the timer must then call the scheduler to allow the higher priority task to preempt τ_i. Notice that, once a timer has been set, other additional activations before the timeout will not prolong the timeout any further.

Finally, cooperative scheduling does not need special kernel support, but it requires the programmer to insert in each preemption point a primitive that calls the scheduler, so enabling pending high-priority tasks to preempt the running task. As a last remark, note that the fixed preemption point model can also be adopted to model electrical loads of a distributed smart grid, where power appliances can be interrupted only at predefined points [32].

B. Predictability

As observed in Section I, the runtime overhead introduced by the preemption mechanism depends on the specific point where the preemption takes place. Therefore, a method that allows predicting where a task is going to be preempted simplifies the estimation of preemption costs, permitting a tighter estimation of task WCETs.

Unfortunately, under preemption thresholds, the specific preemption points depend on the actual execution of the running task and on the arrival time of high-priority tasks, hence it is practically impossible to predict the exact location where a task is going to be preempted.

Under deferred preemptions (floating model), the position of nonpreemptive regions is not specified in the model, thus they are considered to be unknown. In the activation-triggered model, instead, the time at which the running task will be preempted is set q_i units of time after the arrival time of a higher priority task. Hence, the preemption position depends on the actual execution of the running task and on the arrival time of the higher priority task. Therefore, it can hardly be predicted offline.

On the contrary, under fixed preemption points, the locations where preemptions may occur are explicitly defined by the programmer at design time, hence the corresponding overhead can be estimated more precisely by timing analysis tools. For instance, Bertogna *et al.* [33] presented an algorithm for selecting the preemption points that minimize the overall preemption cost without compromising the feasibility of the task set.

C. *Effectiveness*

The effectiveness of an algorithm that limits preemptions can be evaluated either in terms of schedulability or by the number of preemptions. As long as schedulability is concerned, all the limited preemptive methods (under fixed priorities) dominate both fully preemptive scheduling and nonpreemptive scheduling, even when preemption cost is neglected. Such a behavior has been clearly illustrated by showing how the sample task set in Table I cannot be scheduled by fully preemptive and nonpreemptive Deadline Monotonic, whereas it is schedulable using any limited preemptive algorithm. This property will be also evaluated by simulation in the next section, using more quantitative data.

The number of preemptions each task can experience depends of different parameters. Under preemption thresholds, a task τ_i can only be preempted by tasks with priority greater than its threshold θ_i. Hence, if preemption cost is neglected, an upper bound ν_i on the number of preemptions τ_i can experience can be computed by counting the number of activations of tasks with priority higher than θ_i occurring in $[0, R_i]$ that is

$$\nu_i = \sum_{h.P_h > \theta_i} \left\lceil \frac{R_i}{T_h} \right\rceil . \qquad (29)$$

This is an upper bound because each higher priority arrival is counted as a different preemption, even when multiple arrivals cause a single preemption.

Under deferred preemption, the number of preemptions occurring on τ_i can be upper bounded using the nonpreemptive interval q_i specified for the task. If preemption cost is neglected, we simply have

$$\nu_i = \left\lceil \frac{C_i}{q_i} \right\rceil - 1.$$

This is a pessimistic estimation since a task τ_i is assumed to be preempted after every interval of length q_i, even in the absence of higher priority jobs. In this case, a better upper bound can be derived from (29), by replacing θ_i with P_i. Note that when preemption cost is not negligible, the derived upper bounds are not applicable, since task computation times also depend on the number of preemptions, leading to a circular dependency, as shown by Yao *et al.* [34].

Under cooperative scheduling, the number of preemptions can be easily upper bounded by the minimum between the number of effective preemption points inserted in the task code and the number of higher priority jobs activated during the response time of the considered task.

IX. Simulation Experiments

This section presents a set of simulation results performed on randomly generated task sets, with the objective of evaluating the effects of the different scheduling approaches on the number of preemptions and the system schedulability. Specific tests have been carried out to evaluate how schedulability is affected by the size of nonpreemptive regions and by the preemption cost. The aforementioned algorithms have been considered in the comparison, all executed under the Deadline Monotonic priority assignment.

Each task set was generated as follows. The UUniFast [35] algorithm was used to generate a set of n periodic tasks with total utilization equal to a desired value U. Then, for each task τ_i, its computation time C_i was generated as a random integer uniformly distributed in the interval [100, 500], and its period T_i was computed as $T_i = C_i/U_i$. The relative deadline D_i was generated as a random integer uniformly distributed in the range $[C_i + \alpha \cdot (T_i - C_i), T_i]$, with $\alpha = 0.5$.

To reduce preemptions as mush as possible, in the PTS algorithm, threshold priorities were set at the highest possible value using the method described in Section V-B. Similarly, in both DPS and FPP, the length of nonpreemptive regions was set at the highest possible value to keep the task set feasible, using the methods illustrated in Section V-B and Section VII-B, respectively.

In the rest of this section, three sets of experiments are presented. the first set is aimed at evaluating how the number of preemptions is affected by different parameters; the second set evaluates the schedulability level in the ideal case of zero peremption cost, whereas the third set compares the feasibility level in the presence of non-negligible cost.

A. *Number of Preemptions*

The first set of experiments was carried out to monitor the total number of preemptions generated by the different algorithms on a periodic task set within a simulation window of $5 \cdot 10^7$ units of time. In particular, each value shown in the graphs plots the average over 1000 runs. To make the comparison fair, only preemptively feasible task sets were considered and the preemption cost was assumed to be zero. In this set of experiments, the nonpreemptive scheduling algorithm (NPS) is not reported, since the number of preemptions is always zero, for any utilization. Such a great performance of NPS, however, is compensated by a poor schedulability level, which is better evaluated in the second set of experiments. The curve for deferred preemption scheduling (DPS) corresponds to the activation-triggered model. We did not include the floating model because in this model no information is provided on the minimum length and position of the nonpreemptive regions. The number of preemptions is therefore the same as in the fully preemptive case (FPS).

Fig. 7 shows how the performance of the various algorithms varies as a function of the task set utilization, for task sets composed of $n = 10$ tasks. As clear from the graphs, the use of nonpreemptive regions, either fixed (FPP) or not (DPS), allows achieving a higher reduction with respect to preemption thresholds, especially for task set utilizations greater than 70%. Note that in all the graphs related to this experiment, DPS performs slightly better than FPP. This can be explained considering that, when preemption points are fixed, high-priority jobs arriving slightly before and after a preemption point generate two dis-

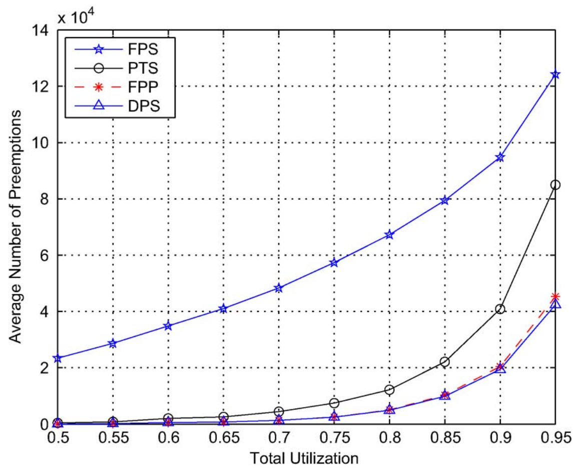


Fig. 7. Average number of preemptions versus utilization when $n = 10$.

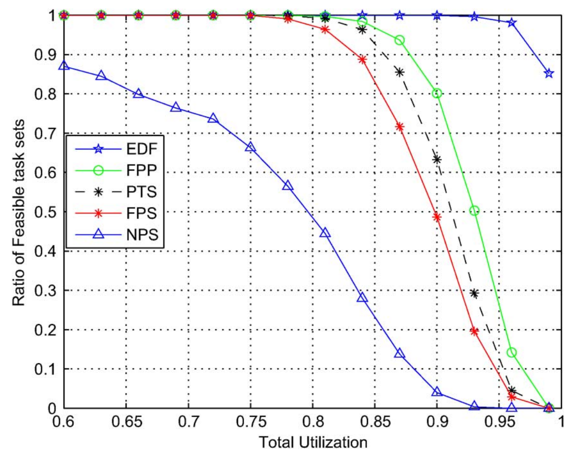


Fig. 9. Feasible ratio versus utilization when $n = 10$.

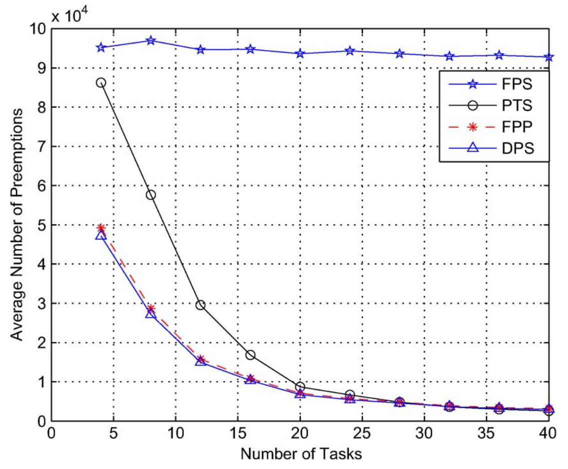


Fig. 8. Average number of preemptions versus number of tasks when $U = 0.9$.

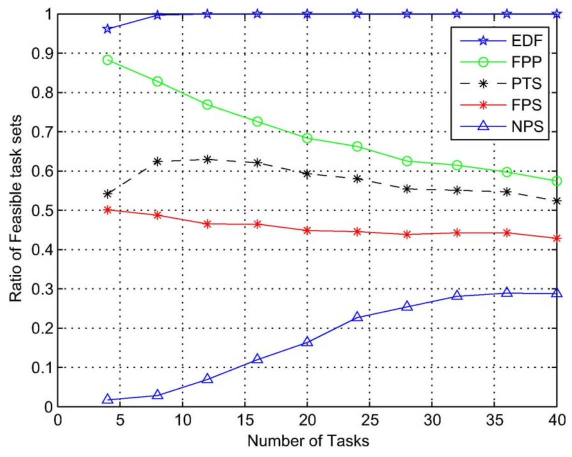


Fig. 10. Feasible ratio as a function of n, when $U = 0.9$.

tinct preemptions (although deferred), whereas under DPS the first arrival always triggers a nonpreemptive interval of length Q, which prevents other subsequent arrivals to generate additional preemptions. In most practical cases, however, such a performance difference is quite negligible, hence FPP is still to be preferred against DPS for the reasons expressed in the previous sections.

Fig. 8 shows the average number of preemptions as a function of the number of tasks, when $U = 0.9$. Note that preemptions rapidly decrease with n for all the algorithms. This is due to the fact that, for a given utilization, large task sets are characterized by tasks with smaller computation times, which have less chance to be preempted. For task sets with $n < 20$, however, both FPP and DPS lead to a significant reduction with respect to PTS.

B. Schedulability With Zero Preemption Cost

The second set of experiments was carried out to test the impact of the various algorithms on the task set schedulability, which has been verified using the feasibility tests reported in this paper, assuming zero preemption cost. The performance of the algorithms was evaluated by comparing the feasible ratio, calculated as the number of feasible task sets divided by the total number of generated sets. In each experiment, 5000 task sets were randomly generated for each parameter configuration. The assumption on preemptive feasibility was removed and the percentage of feasible task sets was monitored as a function of different parameters.

In this set of experiments, DPS is not shown, since its performance is the same as FPS, as mentioned in Section VI-A. On the other hand, fully preemptive EDF [19] has been included in the graphs to evaluate the difference with respect to an optimal solution.

Fig. 9 shows the performance of the various algorithms as a function of the task set utilization, when $n = 10$. It is worth observing that both FPP and PTS improve the schedulability level with respect to FPS, but FPP is able to achieve a larger improvement, especially for higher utilizations $(U > 0.85)$. For example, FPP is able to schedule 30% more task sets than FPS for U around 0.9.

A second experiment has been carried out to test how schedulability is affected by the number of tasks. Here, the total system utilization was set to $U = 0.9$ and the number of tasks was varied from 4 to 40. The results are reported in Fig. 10.

Note that FPP always outperforms all the other fixed priority algorithms, although the improvement decreases for larger task sets. This can be explained observing that a large task set is more likely to have smaller blocking tolerances, due to the higher number of generated deadlines. This phenomenon limits the length of nonpreemptive regions of lower priority tasks, hence FPP has less chance to improve schedulability for large task sets. On the other hand, the performance of NPS increases with n, because larger task sets tend to have smaller computation times, which introduce smaller blocking times in higher priority tasks.

C. Schedulability With Preemption Cost

Considering that FPP is the algorithm that exhibits the best performance with respect to the other fixed priority schemes, a final experiment was carried out to evaluate how the feasibility ratio of FPP is affected by the preemption cost. In this case, however, existing feasibility tests that take preemption costs into account are quite pessimistic, since they count a preemption for each high-priority job arrival. For this reason, in this set of experiments, an approximated feasibility ratio was computed by simulation, considering a task set schedulable if no deadline miss occurred during the entire simulation interval. Even if such a simulation represents just a necessary condition for feasibility, it allows giving a rough estimation of the schedulability performance when preemption cost is taken into account.

Preemption cost has been incorporated into response time analysis by Altmeyer *et al.* [36] to obtain tight bounds on feasibility. However, the approach requires detailed information on the task structure and cache usage, which is not in the scope of this paper.

In the experiment, the length q of the nonpreemptive regions in each task was varied from 0 to $C_{\max}$, (i.e., the longest computation time among the tasks), through a parameter λ varying in [0, 1], such that $q = \lambda C_{\max}$. In this way, FPS and NPS result to be two special cases of FPP, obtained with $\lambda = 0$ and $\lambda = 1$, respectively. Note that, if $q \geq C_i$, task τ_i is entirely executed in nonpreemptive mode. The same q value is used for all tasks in the system in order to vary the length of the nonpreemptive regions in a uniform way for the whole task set. However, a much better schedulability performance could be obtained adopting a different q_i value for each task τ_i, as explained in Section VII. The preemption cost, denoted by γ, was assumed to be a fixed value for each task.

Fig. 11 shows the feasibility ratio achieved by FPP as a function of the task set utilization, for different values of λ in [0, 1]. Here, the task set has $n = 10$ tasks and the preemption cost is $\gamma = 30$. Note that different curves intersect each other, meaning that the relative performance depends on the task set utilization. In particular, using smaller nonpreemptive regions is more efficient for small task set utilizations, when there are less preemptions due to the reduced workload. On the other hand, when the total utilization increases, having longer nonpreemptive regions might help reducing the number of preemptions, reducing the overhead experienced. In the considered configuration, the curve for $\lambda = 0.2$ (i.e., for $q = C_{\max} \cdot 0.2 = 100$) has the best performance until $U = 0.8$, while the curve for $\lambda = 0.4$ (i.e., for $q = C_{\max} \cdot 0.4 = 200$) has a better performance for larger utilizations. It is interesting to note that the curve for fully preemptive scheduling ($\lambda = 0$) has a rapid performance degradation, being the highest one for $U < 0.7$ and the lowest one when $U > 0.85$.

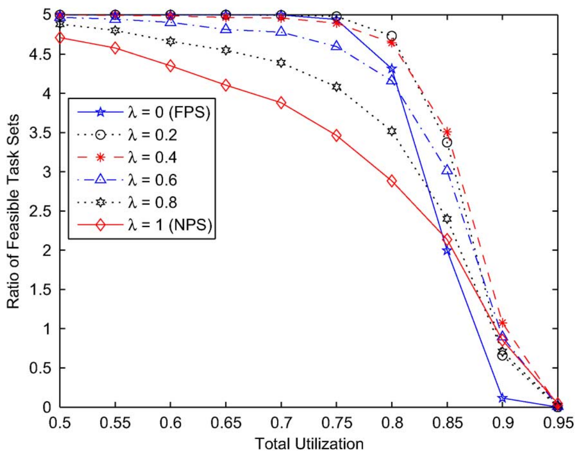


Fig. 11. Feasible ratio versus utilization under different q values, with $n = 10$ and $\gamma = 30$.

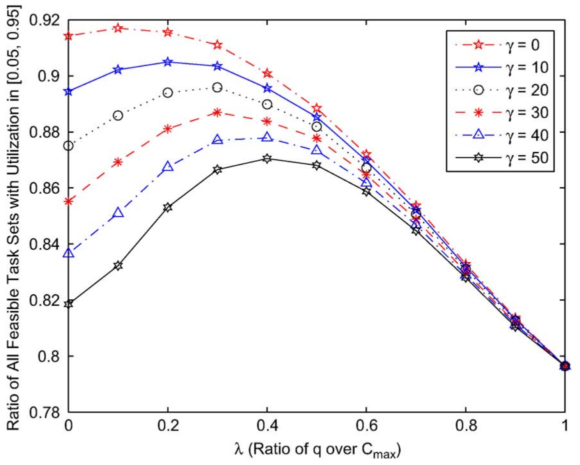


Fig. 12. Overall feasible ratio versus length of nonpreemptive regions.

Finally, Fig. 12 shows how the overall system feasibility, computed for all the task sets generated within the utilization range [0.05, 0.95], varies as a function of λ, from the fully preemptive case ($\lambda = 0$), to the nonpreemptive case ($\lambda = 1$).

Different curves are plotted for different preemption costs ranging from 0 to 50 units of time, with a step of 10. It is worth noting that the highest feasibility ratio is not achieved under fully preemptive scheduling ($\lambda = 0$), even for low preemption costs. Also, note that increasing the preemption cost the highest feasibility ratio is achieved for longer nonpreemptive regions (higher λ). This confirms that limited preemptive scheduling dominates fully preemptive and nonpreemptive scheduling, even when preemption cost is negligible, and becomes more effective for larger preemption costs. Also note that, when λ increases, each task has less chances to be preempted, hence the cost is less relevant and the gap between lines reduces. Eventually, all lines merge at one point, since NPS does suffer from the preemption cost.

X. Conclusion

This paper presented a survey of limited preemptive scheduling algorithms, as methods for increasing the predictability and efficiency of real-time systems. The most relevant result that clearly emerges from the experiments is that, under fixed priority scheduling, any of the considered algorithms dominates both fully preemptive and nonpreemptive scheduling, even when preemption cost is neglected.

As discussed in the previous sections, each specific algorithm for limiting preemptions has advantages and disadvantages. The preemption threshold mechanism has a simple and intuitive interface and can be implemented by introducing a low runtime overhead; however, preemption cost cannot be easily estimated, since the position of each preemption, as well as the overall number of preemptions for each task, cannot be determined offline. Using deferred preemptions, the number of preemptions for each task can be better estimated, but still the position of each preemption cannot be determined offline. Fixed preemption points represents the most predictable solution for estimating preemption costs, since both the number of preemptions and their positions are fixed and known from the task code. Moreover, simulation experiments clearly show that the FPP algorithm is the one generating less preemptions and higher schedulability ratios for any task set parameter configurations. However, FPP requires adding explicit preemption points in the program, hence achieving portability of legacy code is still a challenge for future works.

References

[1] M. Grenier and N. Navet, "Fine-tuning MAC-level protocols for optimized real-time QoS," *IEEE Trans. Ind. Informat.*, vol. 4, no. 1, pp. 6–15, Feb. 2008.

[2] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," *IEEE Trans. Comput.*, vol. 47, no. 6, pp. 700–713, 1998.

[3] H. Ramaprasad and F. Mueller, "Tightening the bounds on feasible preemption points," in *Proc. 27th IEEE Real-Time Syst. Symp. (RTSS'06)*, Rio de Janeiro, Brazil, Dec. 5–8, 2006, pp. 212–224.

[4] H. Ramaprasad and F. Mueller, "Bounding worst-case response time for tasks with non-preemptive regions," in *Proc. Real-Time Embedded Technol. Appl. Symp. (RTAS'08)*, St. Louis, MO, Apr. 22–24, 2008, pp. 58–67.

[5] H. Ramaprasad and F. Mueller, "Tightening the bounds on feasible preemptions," *ACM Trans. Embedded Comput. Syst.*, vol. 10, no. 2, pp. 1–34, Dec. 2010.

[6] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols. An approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, Sep. 1990.

[7] T. P. Baker, "Stack-based scheduling for realtime processes," *Real-Time Syst.*, vol. 3, no. 1, pp. 67–99, Apr. 1991.

[8] G. Buttazzo and A. Cervin, "Comparative assessment and evaluation of jitter control methods," in *Proc. 15th Int. Conf. Real-Time and Network Syst. (RTNS'07)*, Nancy, France, Mar. 29–30, 2007, pp. 137–144.

[9] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo, "A new kernel approach for modular real-time systems development," in *Proc. 13th IEEE Euromicro Conf. Real-Time Syst. (ECRTS'01)*, Delft, The Netherlands, Jun. 13–15, 2001, pp. 199–206.

[10] R. Marau, P. Leite, M. Velasco, P. Marti, L. Almeida, P. Pedreiras, and J. Fuertes, "Performing flexible control on low-cost microcontrollers using a minimal real-time kernel," *IEEE Trans. Ind. Informat.*, vol. 4, no. 2, pp. 125–133, May 2008.

[11] B. D. Bui, M. Caccamo, L. Sha, and J. Martinez, "Impact of cache partitioning on multi-tasking real time embedded systems," in *IEEE Proc. 14th Int. Conf. Embedded Real-Time Comput. Syst. Appl. (RTCSA'08)*, Kaohsiung, Taiwan, Aug. 25–27, 2008, pp. 101–110.

[12] S. Altmeyer and G. Gebhard, "WCET analysis for preemptive scheduling," in *Proc. 8th Int. Workshop Worst-Case Execution Time (WCET) Analysis*, Prague, Czech Republic, Jul. 2008, pp. 105–112.

[13] G. Gebhard and S. Altmeyer, "Optimal task placement to improve cache performance," in *Proc. 7th ACM-IEEE Int. Conf. Embedded Softw. (EMSOFT 07)*, Salzburg, Austria, Oct. 1–3, 2007, pp. 259–268.

[14] C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch," in *Proc. ACM Workshop Experimental Comput. Sci. (ExpCS'07)*, San Diego, CA, Jun. 13–14, 2007.

[15] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *Proc. 6th IEEE Int. Conf. Real-Time Comput. Syst. Appl. (RTCSA'99)*, Hong Kong, China, Dec. 13–15, 1999, pp. 328–335.

[16] S. Baruah, "The limited-preemption uniprocessor scheduling of sporadic task systems," in *Proc. 17th Euromicro Conf. Real-Time Syst. (ECRTS'05)*, Palma de Mallorca, Balearic Islands, Spain, Jul. 6–8, 2005, pp. 137–144.

[17] A. Burns, S. Son, Ed., "Preemptive priority based scheduling. An appropriate engineering approach," *Adv. Real-Time Syst.*, pp. 225–248, 1994.

[18] R. J. Bril, J. J. Lukkien, and W. F. J. Verhaegh, "Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited," in *Proc. 19th Euromicro Conf. Real-Time Syst. (ECRTS'07)*, Pisa, Italy, Jul. 4–6, 2007, pp. 269–279.

[19] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. Assoc. Comput. Mach.*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[20] 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 Syst.*, vol. 42, no. 1–3, pp. 63–119, 2009.

[21] J. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic real-time tasks," *Perform. Eval.*, vol. 2, no. 4, pp. 237–250, 1982.

[22] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller area network (CAN) schedulability analysis. Refuted, revisited and revised," *Real-Time Syst.*, vol. 35, no. 3, pp. 239–272, 2007.

[23] G. Yao, G. Buttazzo, and M. Bertogna, "Feasibility analysis under fixed priority scheduling with fixed preemption points," in *Proc. 16th IEEE Int. Conf. Embedded Real-Time Comput. Syst. Appl. (RTCSA'10)*, Macau, China, Aug. 23–25, 2010, pp. 71–80.

[24] U. Keskin, R. Bril, and J. Lukkien, "Exact response-time analysis for fixed-priority preemption-threshold scheduling," in *Work-in-Progress Session 15th Int. Conf. Emerging Technol. Factory Autom. (ETFA'10)*, Bilbao, Spain, Sep. 13–16, 2010.

[25] J. Regehr, "Scheduling tasks with mixed preemption relations for robustness to timing faults," in *Proc. 23rd IEEE Real-Time Syst. Symp. (RTSS'02)*, Austin, Texas, Dec. 3–5, 2002, pp. 315–326.

[26] M. Saksena and Y. Wang, "Scalable real-time system design using preemption thresholds," in *Proc. 21st IEEE Real-Time Syst. Symp. (RTSS'00)*, Orlando, FL, Nov. 27–30, 2000, pp. 25–34.

[27] M. Bertogna and S. Baruah, "Limited preemption EDF scheduling of sporadic task systems," *IEEE Trans. Ind. Informat.*, vol. 6, no. 4, pp. 579–591, Nov. 2010.

[28] M. Short, "Improved schedulability analysis of implicit deadline tasks under limited preemption edf scheduling," in *Proc. 16th IEEE Conf. Emerging Technol. Factory Autom. (ETFA'11)*, Sep. 2011, pp. 1–8.

[29] G. Yao, G. Buttazzo, and M. Bertogna, "Bounding the maximum length of non-preemptive regions under fixed priority scheduling," in *Proc. 15th IEEE Int. Conf. Embedded Real-Time Comput. Syst. Appl. (RTCSA'09)*, Beijing, China, Aug. 24–26, 2009, pp. 351–360.

[30] E. Bini and G. C. Buttazzo, "Schedulability analysis of periodic fixed priority systems," *IEEE Trans. Comput.*, vol. 53, no. 11, pp. 1462–1473, 2004.

[31] M. Bertogna, G. Buttazzo, and G. Yao, "Improving feasibility of fixed priority tasks using non-preemptive regions," in *Proc. 32nd IEEE Real-Time Syst. Symp. (RTSS 2011)*, Vienna, Austria, Nov. 30–Dec. 2 2011, pp. 251–260.

[32] T. Facchinetti and M. D. Vedova, "Real-time modeling for direct load control in cyber-physical power systems," *IEEE Trans. Ind. Informat.*, vol. 7, no. 4, pp. 689–698, Nov. 2011.

[33] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazzo, "Optimal selection of preemption points to minimize preemption overhead," in *Proc. 23rd Euromicro Conf. Real-Time Syst. (ECRTS'11)*, Porto, Portugal, Jul. 6–8, 2011, pp. 217–227.

[34] G. Yao, G. Buttazzo, and M. Bertogna, "Comparative evaluation of limited preemptive methods," in *Proc. 15th IEEE Int. Conf. Emerging Techonol. Factory Autom. (ETFA'10)*, Bilbao, Spain, Sep. 13–16, 2010, pp. 1–8.

[35] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Syst.*, vol. 30, no. 1–2, pp. 129–154, 2005.

[36] S. Altmeyer, R. Davis, and C. Maiza, "Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems," in *Proc. 32nd IEEE Real-Time Syst. Symp. (RTSS.11)*, Vienna, Austria, Nov. 30–Dec. 2 2011.

Giorgio C. Buttazzo (SM'05–F'12) graduated with a Degree in electronic engineering at the University of Pisa, Pisa, Italy, in 1985, the M.S. degree in computer science from the University of Pennsylvania, Philadelphia, in 1987, and the Ph.D. degree in computer engineering from the Scuola Superiore Sant'Anna of Pisa, Pisa, in 1991.

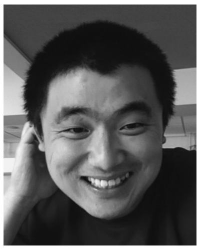
He is a Full Professor of Computer Engineering at the Scuola Superiore Sant'Anna of Pisa. From 1987 to 1988, he worked on active perception and real-time control at the G.R.A.S.P. Laboratory, University of Pennsylvania. He has authored 6 books on real-time systems and over 200 papers in the field of real-time systems, scheduling algorithms, overload management, robotics, and neural networks.

Prof. Buttazzo has been Program Chair and General Chair of the major international conferences on real-time systems. He is Editor-in-Chief of *Real-Time Systems* (Springer), Associate Editor of the IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, and Chair of the IEEE Technical Committee on Real-Time Systems.

Marko Bertogna (SM'11) graduated (summa cum laude) in Telecommunications Engineering at the University of Bologna in 2002. He received the Ph.D. degree in computer engineering from Scuola Superiore Sant'Anna of Pisa, Pisa, Italy, in 2008.

He is an Assistant Professor at the University of Modena and Reggio Emilia, Italy. In 2006, he visited the University of North Carolina at Chapel Hill, working with Prof. S. Baruah on scheduling algorithms for single and multicore real-time systems. His research interests include scheduling and schedulability analysis of real-time multiprocessor systems, protocols for the exclusive access to shared resources, resource reservation algorithms, and reconfigurable devices. He has authored over 30 papers in international conferences and journals in the field of real-time systems.

Prof. Bertogna received four Best Paper Awards.



Gang Yao received the B.E. and M.E. degrees from Tsinghua University, Beijing, China, in 2003 and 2006, respectively, and the Ph.D. degree in computer engineering from the Scuola Superiore SantAnna of Pisa, Pisa, Italy, in 2010.

He is a Postdoctoral Research Collaborator at the University of Illinois at Urbana–Champaign. His interests include real-time scheduling algorithms, safety-critical systems, and shared resource protocols.