PREDATOR

Design for Predictability and Efficiency

Contract no. FP7-ICT-216008

Deliverable No. 3.4

Analysis of preemptive and non-preemptive scheduling

SEVENTH FRAMEWORK PROGRAMME

A project funded under the European Community's 7th Framework Programme

Project ref. no.	FP7-ICT-216008
Project title	Design for Predictability and Efficiency

Contractual date of delivery	Project Month No. 36 - Date: January 31st, 2011
Actual date of delivery	Project Month No. 36 - Date: January 31st, 2011
Deliverable number	D3.4
Deliverable title	Analysis of preemptive and non-preemptive scheduling
Type	Report
Status/version	v1
Number of pages incl. Cover sheet	32
WP contributing to the deliverable	WP3
Task responsible	T3.3
Lead Beneficiary	SSSA
Author(s)	Giorgio Buttazzo, Marko Bertogna, Gang Yao
EC Project Officer	Joao Paulo de Sousa
Keywords	Real-Time Scheduling, Interrupt handling, Aperiodic servers

Contents

Analysis of preemptive and non-preemptive scheduling

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

Abstract

The question whether preemptive systems are better than non-preemptive systems has been debated for a long time, but only partial answers have been provided in the real-time literature and still some issues remain open. In fact, each approach has advantages and disadvantages, and no one dominates the other when both predictability and efficiency have to be taken into account in the system design. In this deliverable, we revise and compare some existing approaches for reducing preemptions and propose an efficient method for minimizing preemption costs by removing unnecessary preemptions while preserving system schedulability.

1 Introduction

Preemption is a key factor in real-time scheduling algorithms, since it allows the operating system to immediately allocate the processor to incoming tasks with higher priority. 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.

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 non-preemptive discipline is that it introduces an additional blocking factor in higher priority tasks, so reducing schedulability. On the other hand, however, there are several advantages to be considered when adopting a non-preemptive scheduler. In particular, the following issues should be taken into account.

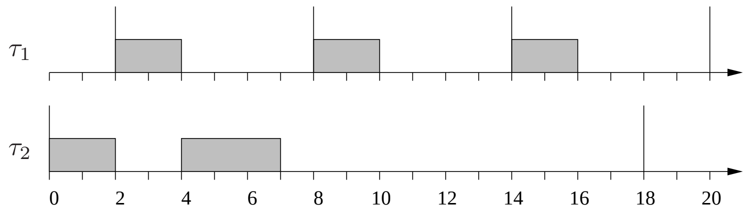
- In many practical situations, such as I/O scheduling or communication in a shared medium, either preemption is impossible or prohibitively expensive.
- Preemption destroys program locality, increasing the runtime overhead due to cache misses and pre-fetch mechanisms. As a consequence, worst-case execution times (WCETs) are more difficult to characterize and predict [28, 36, 35, 34].
- The mutual exclusion problem is trivial in non-preemptive scheduling, which naturally guarantees the exclusive access to shared resources. On the contrary, to avoid unbounded priority inversion, preemptive scheduling requires the implementation of specific concurrency control protocols for accessing shared resources, such as Priority Inheritance, Priority Ceiling [39] or Stack Resource Policy [4], which introduce additional overhead and complexity.
- In control applications, the input-output delay and jitter are minimized for all tasks when using a non-preemptive scheduling discipline, since the interval between start time and finishing time is always equal to the task computation time [17]. This simplifies control techniques for delay compensation at design time.
- Non-preemptive execution allows using stack sharing techniques [4] to save memory space in small embedded systems with stringent memory constraints [21].

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:

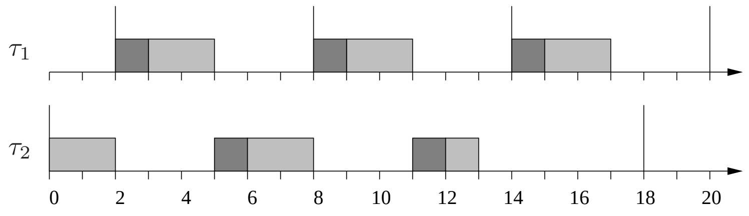
1. *Scheduling cost*. It is the time taken by the scheduling algorithm to suspend the running task, insert it into the ready queue, switch the context, and dispatch the new incoming task.
2. *Pipeline cost*. It accounts for the time taken to flush the processor pipeline when the task is interrupted and the time taken to refill the pipeline when the task is resumed.
3. *Cache-related cost*. It is the time taken to reload the cache lines evicted by the preempting task. This time depends on the specific point in which preemption occurs and on the number of preemptions experienced by the task [1, 23]. Bui et al. [14] 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%.
4. *Bus-related cost*. It is extra bus interference for accessing the RAM due to the additional cache misses caused by preemption.

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 when preemption takes place [1, 23, 31].

The total increase of the worst-case execution time 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. Figure 1(a) shows a simple example in which neglecting preemption cost task τ_2 experiences a single preemption. However, when taking preemption cost into account, τ_2's WCET becomes higher, and hence τ_2 experiences additional preemptions, which in turn increase its WCET. In Figure 1(a) the architecture cost due to preemption is represented by dark gray areas.



(a) Schedule without preemption cost.

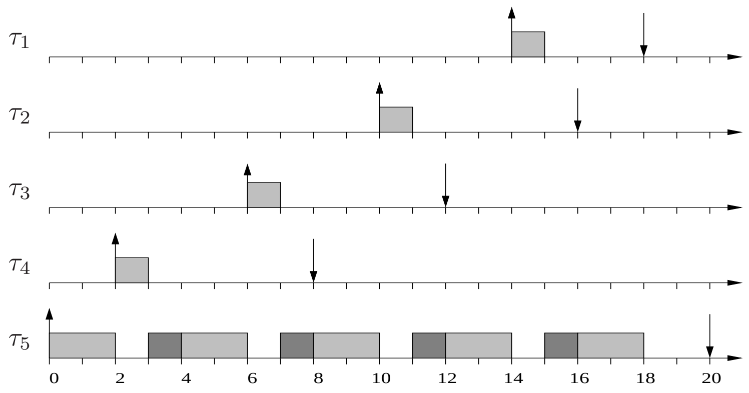


(b) Schedule with preemption cost.

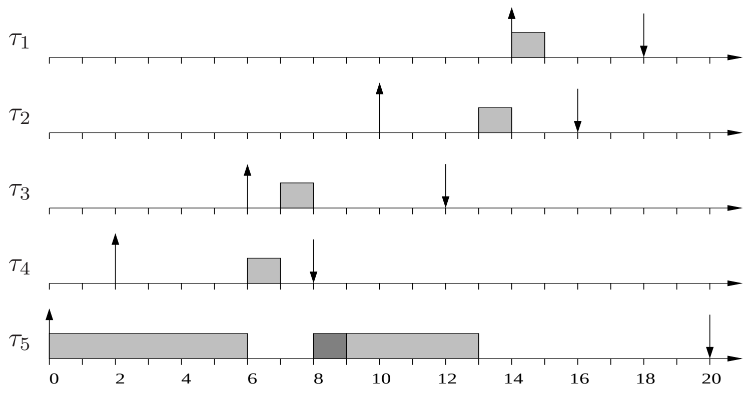
Figure 1. Task τ_2 experiences a single preemption when preemption cost is neglected, and two preemptions when preemption cost is taken into account.

Some methods for estimating the number of preemptions have been proposed [19, 47], but they are restricted to the fully preemptive case and do not consider such a circular dependency.

Often, preemption is considered a pre-requisite to meet timing requirement in real-time system design; however, in most cases, a fully preemptive scheduler produces many unnecessary preemptions. Figure 2(a) illustrates an example in which, under fully preemptive scheduling, task τ_5 is preempted four times. As a consequence, the WCET of τ_5 is substantially inflated by the architecture related cost (represented by dark gray areas), causing a response time equal to $R_5 = 18$. However, as shown in Figure 2(b), only one preemption is really necessary to guarantee the schedulability of the task set, reducing the WCET of τ_5 from 14 to 11 units of time, and its response time from 18 to 13 units.



(a) τ_5 is preempted 4 times.



(b) Only one preemption is really necessary for τ_5.

Figure 2. Fully Preemptive scheduling can generate several preemptions (a), although only a few of them a really necessary to guarantee the schedulability of the task set (b).

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*. According to this approach, proposed by Wang and Saksena [43], a task is allowed to disable preemption up to a specified priority level, which is 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*. According to this method, each task τ_i specifies the longest interval q_i that can be executed non-preemptively. Depending on how non preemptive regions are implemented, this model can come in two slightly different flavors:

  1. *Floating model*. In this model, non-preemptive regions are defined by the programmer by inserting specific primitives in the task code that disable and enable preemption. Since the start time of each region is not specified in the model, non-preemptive regions cannot be identified off line and, for the sake of the analysis, are considered to be "floating" in the code, with a duration $\delta_{i,k} \leq q_i$.
  2. *Time-triggered model*. In this model, non-preemptive regions are triggered by the arrival of a higher priority task and enforced by a timer to last 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. After the timeout, a new high-priority arrival can trigger another non-preemptive region.

- *Task splitting*. Using this approach, each task implicitly executes in non-preemptive mode and preemption is enabled by inserting a specific system call in the code, so allowing preemption to take place only in predefined locations, called *preemption points*. In this way, a task is divided into a number of non-preemptive 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.

To better understand the different limited preemptive approaches, the task set reported in Table 1 will be used as a common example.

	C_i	T_i	D_i
τ_1	1	6	4
τ_2	3	10	8
τ_3	6	18	12

Table 1. Parameters of a sample task set with relative deadlines less than periods.

Figure 3 illustrates the schedule produced by Deadline Monotonic (in fully preemptive mode) on the task set of Table 1. Notice that the task set is not schedulable, since task τ_3 misses its deadline.

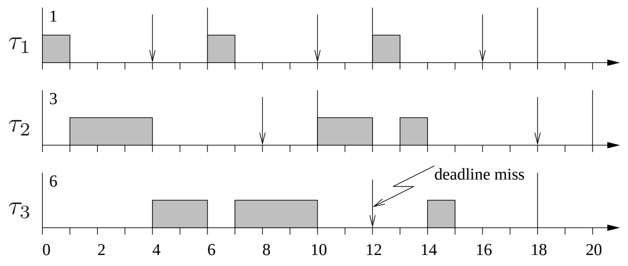


Figure 3. Schedule produced by Deadline Monotonic (in fully preemptive mode) on the task set of Table 1.

1.1 Related work

Most work on non-preemptive scheduling has typically focused on single-job models, where tasks are invoked only once, must be completed before a deadline and can have precedence relations [20, 22]. Non-preemptive tasks were considered in the Spring Kernel [40], where a heuristic algorithm was used to find a feasible schedule or reduce the number of deadline misses. A more general characterization of periodic tasks has been considered in [27, 30]. In this model, tasks may have a deadline smaller than or equal to the next release time. For this more general model, Mok [33] has shown that the problem of deciding schedulability of a set of periodic tasks with mutually exclusive sections of code is NP-hard.

Jeffay et al. [26] showed that non-preemptive scheduling of concrete periodic tasks[1] is NP-hard in the strong sense. George et al. [24] provided comprehensive feasibility analysis on non-preemptive scheduling, however, the authors assumed either a completely non-preemptive or a fully preemptive model. Davis et al. [18] considered typical applications of non-preemptive fixed priority scheduling on a CAN bus, and presented the analysis to bound worst-case response times of real-time messages.

Fixed priority scheduling with deferred preemptions, allowed only at some predefined points inside the task code, has been proposed and investigated by Burns [15], who however did not address the problem of computing the maximum length of non-preemptive chunks. Bril et al. [13] further improved the response time analysis under

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

this model. The authors identified a critical situation that may occur in the presence of non-preemptive regions, deriving the analysis to take such a phenomenon into account. In particular, in certain situations, the execution of the last non-preemptive chunk of a task τ_i can delay the execution of one or some higher priority tasks, which can later interfere with the subsequent invocations of τ_i. Identifying such a situation, later referred to as *self-pushing* phenomenon, requires a more complex test, since the analysis cannot be limited to the first job of each task, but it must be performed on multiple task instances within a certain period.

Under the deferred preemption model, Baruah [6] computed the longest non-preemptive interval for each task that does not jeopardize the schedulability of the task set under EDF. Yao et al. [45] addressed the same problem, but under fixed priorities. Later, Yao et al. [44] extended the analysis under cooperative scheduling and presented a comparative study to evaluate the impact on schedulability of different limited preemptive methods [46].

When taking preemption overhead into account, the schedulability analysis becomes more complex, because cache-related preemption delays (CRPDs) significantly increase worst-case execution times [28, 42], which in turn affect the total number of preemptions [36]. Under cooperative scheduling, however, the negative influence of CRPDs can be alleviated by appropriately selecting the potential preemption points. In [8], a method was proposed to select the preemption points, under the assumption of a fixed preemption cost at each preemption point.

Techniques to estimate the cache-related preemption delays have been proposed in [25, 28]. However, most research results considered only a single task in the analysis. A method to incorporate the effect of instruction cache on response time analysis has been proposed in [16]. Only recently, some more general frameworks [36, 41] have been proposed to deal with multi-task real-time systems. Finally, a partial preemptive model [35] has been proposed to consider preemption cost under limited preemption, however each task can only have a single non-preemptive region.

1.2 Terminology and notation

We consider a set of n periodic and sporadic real-time tasks to be scheduled on a single processor. Each task τ_i is characterized by a worst-case execution time (WCET) C_i, a relative deadline D_i, and a period (or minimum inter-arrival time) T_i. A constrained deadline model is adopted here, so D_i is assumed to be less than or equal to T_i. For scheduling purposes, 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., $\forall i \mid 1 \leq i < n : P_i > P_{i+1}$. Additional terminology will be introduced below for each specific method.

1.3 Structure of the report

The rest of this report is organized as follows: Section 2 briefly recalls the main theoretical results derived in the literature to verify the feasibility of a task set under preemptive scheduling. Section 3 discusses how to modify the schedulability analysis when tasks are executed in a non-preemptive fashion. The three limited preemptive approaches, namely preemption thresholds, deferred preemptions, and task splitting, are analyzed in Sections 4, 5, and 6, respectively. Section 7 illustrates an algorithm for selecting the preemption points that minimize the overall preemption cost for each task. Section 8 provides a comparison of the approaches, and Section 9 discusses the results and states the conclusions.

2 Theoretical background

This section summarizes the results that have been derived in the literature for a set of fully preemptive periodic tasks. For tasks with relative deadlines equal to periods, Liu and Layland [32] proved the following theorems.

Theorem 1 (Liu and Layland, 1973)**.** *A set of n fully preemptive periodic tasks with relative deadlines equal to periods can be scheduled under the Rate Monotonic (RM) algorithm (which assigns higher priorities to tasks with higher activation rates) if*

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1). \tag{1}$$

Theorem 2 (Liu and Layland, 1973)**.** *A set of n fully preemptive periodic tasks with relative deadlines equal to periods can be scheduled under the Earliest Deadline First (EDF) algorithm (which assigns higher priorities to tasks with earlier absolute deadlines) if and only if*

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1. \tag{2}$$

Bini et al. [9, 10] derived a tighter bound for Rate Monotonic (the Hyperbolic Bound), deriving the following test:

Theorem 3 (Bini et al., 2001)**.** *A set of n fully preemptive periodic tasks with relative deadlines equal to periods can be scheduled under the Rate Monotonic algorithm if*

$$\prod_{i=1}^{n} \left(\frac{C_i}{T_i} + 1 \right) \leq 2. \tag{3}$$

The schedulability of tasks with fixed priorities and relative deadlines less than or equal to periods can be verified by using the Response Time Analysis (RTA), proposed by Audsley et al. [3, 2].

Theorem 4 (Audsley et al., 1992)**.** *A set of n fully preemptive periodic tasks with relative deadlines equal to periods can be scheduled under the Deadline Monotonic algorithm (which assigns higher priorities to tasks with smaller relative deadlines) if and only if*

$$\forall i = 1, \ldots, n \quad R_i \leq D_i \tag{4}$$

where the response time R_i of task τ_i can be computed by the following recurrent relation:

$$\begin{cases} R_i^{(0)} = C_i \\ R_i^{(s)} = C_i + \displaystyle\sum_{h:P_h > P_i} \left\lceil \frac{R_i^{(s-1)}}{T_h} \right\rceil C_h. \end{cases} \tag{5}$$

In particular, R_i is the smallest value for which $R_i^{(s)} = R_i^{(s-1)}$.

Another necessary and sufficient test for checking the schedulability of fixed priority systems is the one proposed by Lehoczky, Sha, and Ding [29], which is based on the concept of Level-i workload $W_i(t)$, that is the workload requested in the interval $[0, t]$ by tasks with priority higher than or equal to P_i:

$$W_i(t) = \sum_{h:P_h \geq P_i} \left\lceil \frac{t}{T_h} \right\rceil C_h. \tag{6}$$

Then, the test can be expressed by the following theorem:

Theorem 5 (Lehoczky-Sha-Ding, 1989)**.** *A set of fully preemptive periodic tasks can be scheduled under the Deadline Monotonic algorithm if and only if*

$$\forall i = 1, \ldots, n \quad \exists t \in (0, D_i] : W_i(t) \leq t. \tag{7}$$

Later, Bini and Buttazzo [11] restricted the number of points in which condition (7) has to be checked to the following set:

$$\mathcal{TS}(\tau_i) \doteq \mathcal{P}_{i-1}(D_i) \tag{8}$$

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

$$\begin{cases} \mathcal{P}_0(t) = \{t\} \\ \mathcal{P}_i(t) = \mathcal{P}_{i-1}\left(\left\lfloor \frac{t}{T_i} \right\rfloor T_i \right) \cup \mathcal{P}_{i-1}(t). \end{cases} \tag{9}$$

Thus, the schedulability test can be expressed by the following theorem:

Theorem 6 (Bini and Buttazzo, 2004)**.** *A set of fully preemptive periodic tasks can be scheduled under the Deadline Monotonic algorithm if and only if*

$$\forall i = 1, \ldots, n \quad \exists t \in \mathcal{TS} : W_i(t) \leq t. \tag{10}$$

Under EDF, the schedulability of tasks with relative deadlines less than or equal to periods can be verified using the Processor Demand Criterion (PDC), proposed by Baruah, Rosier, and Howell [7]. The analysis is based on the concept of *Demand Bound Function* $\mathsf{dbf}(t)$, which is the amount of processing time requested by task instances having activation and absolute deadline in $[0, t]$. For a set of synchronous periodic tasks, the demand bound function can be expressed as follows:

$$\mathsf{dbf}(t) = \sum_{i=1}^{n} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor C_i. \tag{11}$$

Then, the schedulability test can be expressed by the following theorem:

Theorem 7. *A set of n synchronous periodic tasks with relative deadlines less than or equal to periods can be scheduled by EDF if and only if*

$$\forall t \in \mathcal{D} \quad \sum_{i=1}^{n} \mathsf{dbf}(t) \leq t. \tag{12}$$

where $\mathcal{D}$ is the set of all absolute deadlines no greater than a certain point in time, given by the minimum between the hyperperiod[2] *H and a bound L^*:*

$$\mathcal{D} = \{d_k \mid d_k \leq \min(L^*, H)\}$$

$$L^* = \frac{\sum_{i=1}^{n} (T_i - D_i) U_i}{1 - U}.$$

2.1 Extension with blocking terms

In the presence of mutually exclusive resources or non preemptive sections of code, tasks can experience additional blocking times that must be taken into account in the analysis. All schedulability tests derived for the preemptive case can be extended to include blocking terms, whose values depend on the specific concurrency control protocol adopted in the schedule. In general, all the extended tests guarantee one task τ_i at the time, by inflating its computation time C_i by the blocking factor B_i. In addition, all the guarantee tests that were necessary and sufficient under preemptive scheduling become only sufficient in the presence of blocking factors, since blocking conditions are derived in worst- case scenarios that differ for each task and could never occur in practice.

Liu and Layland test for Rate Monotonic. A set of periodic tasks with blocking factors and relative deadlines equal to periods can be scheduled by RM if

$$\forall i = 1, \ldots, n \qquad \sum_{h: P_h > P_i} \frac{C_h}{T_h} + \frac{C_i + B_i}{T_i} \leq i(2^{1/i} - 1). \tag{13}$$

Liu and Layland test for EDF. A set of periodic tasks with blocking factors and relative deadlines equal to periods can be scheduled by EDF if

$$\forall i = 1, \ldots, n \qquad \sum_{h: P_h > P_i} \frac{C_h}{T_h} + \frac{C_i + B_i}{T_i} \leq 1. \tag{14}$$

[2]The hyperperiod of a periodic task set is defined as the minimum interval of time after which the sequence of activations repeats itself. It is equal to the least common multiple of the periods.

Hyperbolic Test. Using the Hyperbolic Bound, a task set with blocking factors is schedulable by RM if

$$\forall i = 1, \ldots, n \qquad \prod_{h:P_h > P_i} \left(\frac{C_h}{T_h} + 1 \right) \left(\frac{C_i + B_i}{T_i} + 1 \right) \leq 2. \tag{15}$$

Response Time Analysis. Under blocking conditions, the response time of a generic task τ_i with a fixed priority can be computed by the following recurrent relation:

$$\begin{cases} R_i^{(0)} = B_i + C_i \\ R_i^{(s)} = B_i + C_i + \sum_{h:P_h > P_i} \left\lceil \frac{R_i^{(s-1)}}{T_h} \right\rceil C_h. \end{cases} \tag{16}$$

Workload Analysis. Similarly, using the workload analysis, a task set with blocking factors is schedulable by a fixed priority assignment if

$$\forall i = 1, \ldots, n \quad \exists\, t \in \mathcal{TS} : B_i + W_i(t) \leq t. \tag{17}$$

Processor Demand Criterion. The Processor Demand Criterion in the presence of blocking terms has been extended by Baruah [5], using the concept of *Blocking Function* $B(t)$, defined as the largest amount of time for which a job of some task with relative deadline $\leq t$ may be blocked by a job of some task with relative deadline $> t$.

If δ_{jh} denotes the maximum length of time for which τ_j holds a resource that is also needed by τ_h, the blocking function can be computed as follows:

$$B(t) = \max \{ \delta_{jh} | D_j > t \text{ and } D_h \leq t \}. \tag{18}$$

Then, a task set can be scheduled by EDF if

$$\forall t \in \mathcal{D} \quad B(t) + \mathsf{dbf}(t) \leq t. \tag{19}$$

where $\mathcal{D}$ is the set of all absolute deadlines no greater than a certain point in time, given by the minimum between the hyperperiod H and the following expression:

$$\max \left(D_n, \frac{\sum_{i=1}^{n} (T_i - D_i) U_i}{1 - U} \right).$$

3 Non-preemptive scheduling

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,

$$B_i = \max_{j:P_j < P_i} \{ C_j - 1 \} \tag{20}$$

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 Equation (20) is computed over a larger set of tasks. Figure 4 illustrates the schedule generated by Deadline Monotonic on the task set of Table 1 when preemptions are disabled. With respect to the schedule shown in Figure 3, 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 non preemptive scheduling, the least upper bounds of both RM and EDF 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. For example, the task set illustrated in Figure 5(a) is not feasible under non-preemptive Rate Monotonic scheduling (as well as under non-preemptive EDF), since $C_2 > T_1$, but its utilization can be set arbitrarily low by reducing C_1 and increasing T_2. The same task set is clearly feasible when preemption is enabled, as shown in Figure 5(b).

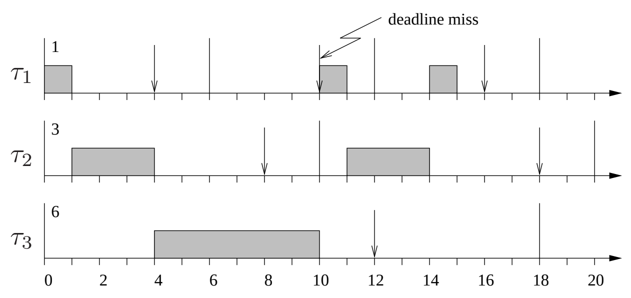


Figure 4. Schedule produced by non-preemptive Deadline Monotonic on the task set of Table 1.

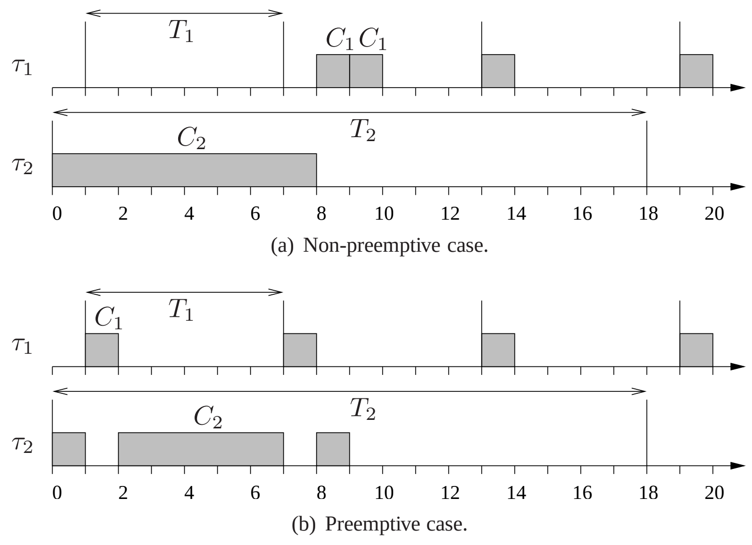


Figure 5. A task set with low utilization that is unfeasible under non-preemptive Rate Monotonic scheduling, and feasible when preemption is enabled.

3.1 Feasibility Analysis

The feasibility analysis of non preemptive task sets is more complex than under fully preemptive scheduling. Bril et al. [13] showed that in non-preemptive 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 Figure 6, where the worst-case response time of τ_3 occurs in its second instance. This kind of a scheduling anomaly, identified as *self-pushing phenomenon*, occurs because a job of τ_3 indirectly pushes (through the non preemptive execution of other tasks) some of its successive jobs, which then experience a higher interference.

The presence of the self-pushing phenomenon in non-preemptive 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 [13]:

Definition 1. *The Level-i Active Period is a continuous interval of time such that tasks with priority $\geq P_i$ are active (ready or running) inside the interval and idle outside it.*

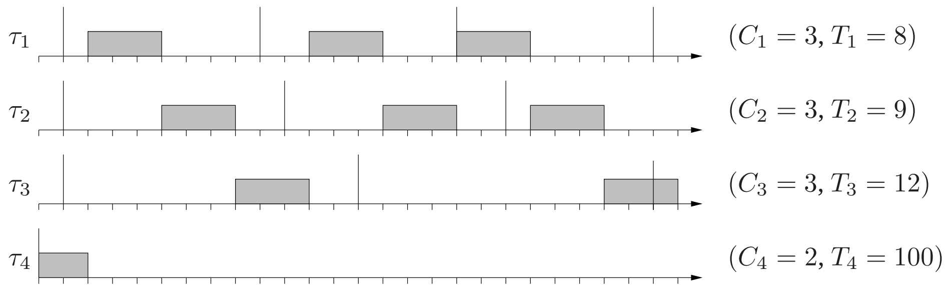


Figure 6. An example of self-pushing phenomenon occurring on task τ_3.

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

$$\begin{cases} L_i^{(0)} = B_i + C_i \\ L_i^{(s)} = B_i + \sum_{h:P_h \geq P_i} \left\lceil \frac{L_i^{(s-1)}}{T_h} \right\rceil C_h. \end{cases} \tag{21}$$

In particular, L_i is the smallest value for which $L_i^{(s)} = L_i^{(s-1)}$.

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:

$$K_i = \left\lceil \frac{L_i}{T_i} \right\rceil. \tag{22}$$

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,

$$s_{i,k} = B_i + (k-1)C_i + \sum_{h:P_h > P_i} \left(\left\lfloor \frac{s_{i,k}}{T_h} \right\rfloor + 1 \right) C_h. \tag{23}$$

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

$$f_{i,k} = s_{i,k} + C_i. \tag{24}$$

Hence, the response time of task τ_i is given by

$$R_i = \max_{k \in [1,K_i]} \{f_{i,k} - (k-1)T_i\}. \tag{25}$$

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

$$\forall i = 1, \ldots, n \quad R_i \leq D_i. \tag{26}$$

Yao, Buttazzo, and Bertogna [44] showed that the analysis of non-preemptive tasks can be reduced to a single job, under specific (but not too restrictive) conditions.

Theorem 8 (Yao, Buttazzo, and Bertogna, 2010)**.** *The worst-case response time of a non-preemptive task occurs in the first job, if the task is activated at its critical instant, and 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.*

In these conditions, the longest relative start time S_i of task τ_i is equal to $s_{i,1}$ and can be computed from Equation (23) for $k = 1$.

$$S_i = B_i + \sum_{h:P_h > P_i} \left(\left\lfloor \frac{S_i}{T_h} \right\rfloor + 1 \right) C_h. \tag{27}$$

Hence, the response time R_i is simply:

$$R_i = S_i + C_i. \tag{28}$$

4 Preemption threshold scheduling (PTS)

According to this model, proposed by Wang and Saksena [43], 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 $P_h > \theta_i$.

Figure 7 illustrates how the threshold is used to raise the priority of a task τ_i during the execution of its k-th job. At the activation time $r_{i,k}$, the priority of a task 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 $P_h > P_i$. 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.

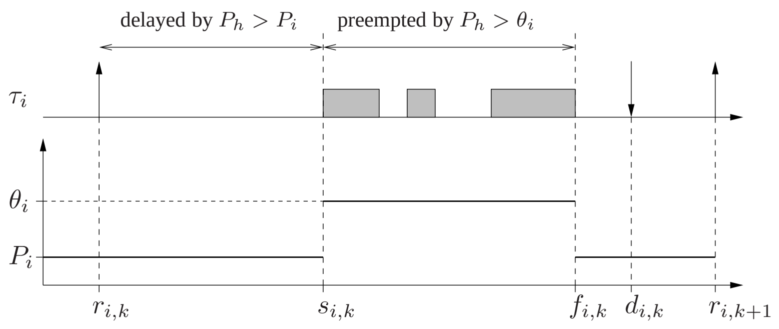


Figure 7. An example of how the threshold is used to raise the priority of a task during execution.

Preemption threshold can be considered as a trade-off between fully preemptive and fully non-preemptive scheduling. Indeed, if each threshold priority is set equal to the task nominal priority, the scheduler behaves like the fully preemptive scheduler; whereas, if all thresholds are set to the maximum priority, the scheduler runs in non-preemptive fashion. Wang and Saksena also showed that, by appropriately setting the thresholds, the system can achieve a higher utilization efficiency, compared with fully preemptive and fully non-preemptive scheduling. For example, assigning the preemption thresholds shown in Table 2, the task set of Table 1 results to be schedulable by Deadline Monotonic, as illustrated in Figure 8.

	P_i	θ_i
τ_1	3	3
τ_2	2	3
τ_3	1	2

Table 2. Preemption thresholds assigned to the tasks of Table 1.

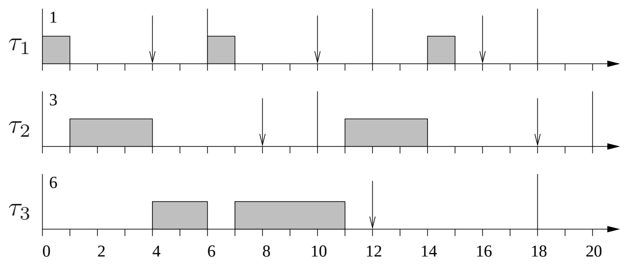


Figure 8. Schedule produced by preemption thresholds for the task set reported in Table 1.

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

4.1 Feasibility Analysis

Under fixed priorities, the feasibility analysis of a task set with preemption thresholds can be performed by the feasibility test derived by Wang and Saksena [43], and later refined by Regehr [37]. 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 having a priority $P_j < P_i$ and a threshold $\theta_j \geq P_i$. 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,

$$B_i = \max_j \{C_j - 1 \mid P_j < P_i \leq \theta_j\} \tag{29}$$

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), as depicted in Figure 7. The analysis must be carried out within the longest Level-i active period L_i, defined by the following recurrent relation:

$$L_i = B_i + \sum_{h:P_h \geq P_i} \left\lceil \frac{L_i}{T_h} \right\rceil C_h. \tag{30}$$

This means that the response time of τ_i must be computed for all jobs $\tau_{i,k}$ $(k = 1, 2, \ldots)$ within the longest busy period. That is, for all $k \in [1, K_i]$, where:

$$K_i = \left\lceil \frac{L_i}{T_i} \right\rceil. \tag{31}$$

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,

$$s_{i,k} = B_i + (k-1)C_i + \sum_{h:P_h > P_i} \left(\left\lfloor \frac{s_{i,k}}{T_h} \right\rfloor + 1 \right) C_h. \tag{32}$$

For the same job $\tau_{i,k}$, 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,

$$f_{i,k} = s_{i,k} + C_i + \sum_{h:P_h > \theta_i} \left(\left\lceil \frac{f_{i,k}}{T_h} \right\rceil - \left(\left\lfloor \frac{s_{i,k}}{T_h} \right\rfloor + 1 \right) \right) C_h. \tag{33}$$

Hence, the response time of task τ_i is given by

$$R_i = \max_{k \in [1, K_i]} \{f_{i,k} - (k-1)T_i\}. \tag{34}$$

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

$$\forall i = 1, \ldots, n \quad R_i \leq D_i. \tag{35}$$

The feasibility analysis under preemption thresholds can also be simplified under the conditions of Theorem 8. In this case, we have that the worst-case start time is

$$S_i = B_i + \sum_{h:P_h > P_i} \left(\left\lfloor \frac{S_i}{T_h} \right\rfloor + 1 \right) C_h \tag{36}$$

and the worst-case response time of task τ_i can be computed as

$$R_i = S_i + C_i + \sum_{h:P_h > \theta_i} \left(\left\lceil \frac{R_i}{T_h} \right\rceil - \left(\left\lfloor \frac{S_i}{T_h} \right\rfloor + 1 \right) \right) C_h. \tag{37}$$

4.2 Selecting preemption thresholds

The example illustrated in Figure 8 shows that a task set unfeasible under both preemptive and non-preemptive scheduling can be feasible under preemption thresholds, for a suitable setting of threshold levels. The algorithm presented in Figure 9 was proposed by Wang and Saksena [43] and allows assigning a set of thresholds to achieve a feasible schedule, if there exists one. Threshold assignment is started from the lowest priority task to the highest priority one, since the schedulability analysis only depends on the thresholds of tasks with lower priority than the current task. While searching the optimal preemption threshold for a specific task, the algorithm stops at the minimum preemption threshold that makes it schedulable. The algorithm assumes that tasks are ordered by decreasing priorities, being τ_1 the highest priority task.

```
Algorithm: Assign Minimum Preemption Thresholds
Input: A task set $\mathcal{T}$ with $\{C_i, T_i, D_i, P_i\}, \forall\tau_i \in \mathcal{T}$
Output: Task set feasibility and $\theta_i, \forall\tau_i \in \mathcal{T}$
// Assumes tasks are ordered by decreasing priorities
(1)  begin
(2)      for ($i := n$ to 1) do          // from the lowest priority task
(3)          $\theta_i = P_i$;
(4)          Compute $R_i$ by Equation (34);
(5)          while ($R_i > D_i$) do          // while not schedulable
(6)              $\theta_i = \theta_i + 1$;          // increase threshold
(7)              if ($\theta_i > P_1$) then          // system not schedulable
(8)                  return (INFEASIBLE);
(9)              end
(10)             Compute $R_i$ by Equation (34);
(11)         end
(12)     end
(13)     return (FEASIBLE);
(14) end
```

Figure 9. Algorithm for assigning the minimum preemption thresholds.

Notice that the algorithm is optimal in the sense that, if there exists a preemption threshold assignment that can make the system schedulable, the algorithm will always find an assignment that ensures schedulability.

Given a task set that is feasible under preemptive scheduling, another interesting problem is to determine the thresholds that limit preemption as much as possible, without jeopardizing the schedulability of the task set. The algorithm shown in Figure 10, proposed by Saksena and Wang [38], tries 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.

5 Deferred Preemptions

According to this method, each task τ_i defines a maximum interval of time q_i in which it can execute non-preemptively. Depending on the specific implementation, the non-preemptive interval can be trigger by the invocation of a system call inserted at the beginning of a non preemptive region (floating model), or by the arrival of a higher priority task (time-triggered model). Under the floating model, preemption is resumed by another system call, inserted at the end of the region (long at most q_i units); whereas, under the time-triggered model, preemption is enabled by a time interrupt after exactly q_i units (unless the task completes earlier).

Since, in both cases, the start times of non-preemptive intervals are assumed to be unknown a priori, non-preemptive regions cannot be identified off line and, for the sake of the analysis, they are considered to occur in the worst possible time (in the sense of schedulability).

```
Algorithm: Assign Maximum Preemption Thresholds
Input: A task set $\mathcal{T}$ with $\{C_i, T_i, D_i, P_i\}, \forall \tau_i \in \mathcal{T}$
Output: Thresholds $\theta_i, \forall \tau_i \in \mathcal{T}$
// Assumes that the task set is preemptively feasible
(1)  begin
(2)      for ($i := 1$ to $n$) do
(3)          $\theta_i = P_i$;
(4)          $k = i$;                          // priority level $k$
(5)          schedulable := TRUE;
(6)          while ((schedulable := TRUE) and ($k > 1$)) do
(7)              $k = k - 1$;                  // go to the higher priority level
(8)              $\theta_i = P_k$;             // set threshold at that level
(9)              Compute $R_k$ by Equation (34);
(10)             if ($R_k > D_k$) then         // system not schedulable
(11)                 schedulable := FALSE;
(12)                 $\theta_i = P_{k+1}$;     // assign the previous priority level
(13)             end
(14)         end
(15)     end
(16) end
```

Figure 10. Algorithm for assigning the maximum preemption thresholds.

For example, considering the same task set of Table 1, assigning $q_2 = 2$ and $q_3 = 1$, the schedule produced by Deadline Monotonic with deferred preemptions is feasible, as illustrated in Figure 11. Dark regions represent intervals executed in non-preemptive mode, started just before the arrival of higher priority tasks.

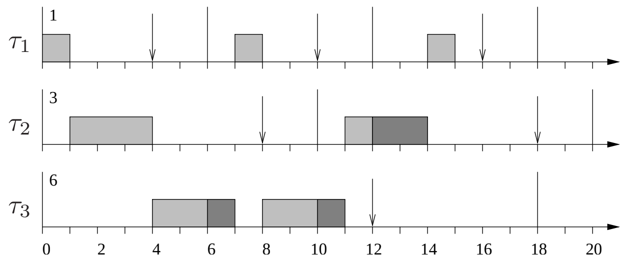


Figure 11. Schedule produced by Deadline Monotonic with deferred preemptions for the task set reported in Table 1, with $q_2 = 2$ and $q_3 = 1$.

5.1 Feasibility Analysis

In the presence of non-preemptive intervals, a task can be blocked when, at its arrival, a lower priority task is running in non-preemptive mode. Since each task can be blocked at most once by a single lower priority task, B_i is equal to the longest non-preemptive interval belonging to tasks with lower priority. That is,

$$B_i = \max_{j:P_j<P_i} \{q_j - 1\}. \tag{38}$$

Then, schedulability can be checked through the response time analysis, by Equation (16), or through the workload analysis, by Equation (17). Note that such an analysis is exact under the floating model. In fact, since non-preemptive regions can have an arbitrary duration no greater than q_i, the worst-case interference on τ_i can actually occur, assuming that τ_i can be preempted an epsilon before its completion.

On the other hand, the analysis is more pessimistic under the time-triggered model, because it does not take advantage of the fact that τ_i cannot be preempted when higher periodic tasks arrive q_i units (o less) before its completion. The advantage of such a pessimism is that the analysis can be limited to the first job of each task.

5.2 Longest non-preemptive interval

When using the deferred preemption method, an interesting problem is to find the longest non-preemptive 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 preemptive periodic tasks that is feasible under a fixed priority assignment, find the longest non-preemptive 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 non-preemptive mode, without violating the schedulability of the original system.

The solution of this problem has been derived by Yao et al.[45] and it is based on the concept of *blocking tolerance* β_i, for a task τ_i, defined as follows:

Definition 2. *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.*

A simple way to compute the blocking tolerance is from the Liu and Layland test, which, in the presence of blocking factors, becomes:

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

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

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

Hence:

$$\beta_i = T_i \left[U_{lub}(i) - \sum_{h:P_h>P_i} \frac{C_h}{T_h} \right]. \tag{39}$$

A more precise bound for β_i can be achieved by using the test, expressed by Equation (17), which leads to the following result:

$$\exists t \in \mathcal{TS}(\tau_i) : B_i \leq \{t - W_i(t)\}.$$

$$B_i \leq \max_{t \in \mathcal{TS}(\tau_i)} \{t - W_i(t)\}.$$

$$\beta_i = \max_{t \in \mathcal{TS}(\tau_i)} \{t - W_i(t)\}. \tag{40}$$

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

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

and, by Equation (38), we can write:

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

This implies that, to achieve feasibility, we must have:

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

Hence, the longest non-preemptive interval Q_i that preserves feasibility for each task τ_i is:

$$Q_i = \min_{k:P_k>P_i} \{\beta_k\}. \tag{41}$$

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 9. *The longest non-preemptive interval Q_i of task τ_i that preserves feasibility can be computed as*

$$Q_i = \begin{cases} \infty & \text{if } i = 1 \\ \min\{Q_{i-1}, \beta_{i-1}\} & \text{otherwise} \end{cases} \tag{42}$$

Proof. The theorem can be proved by noting that

$$\min_{k:P_k>P_i} \{\beta_k\} = \min\{\min_{k:P_k>P_{i-1}} \{\beta_k\}, \beta_{i-1}\}$$

and since from Equation (41)

$$Q_{i-1} = \min_{k:P_k>P_{i-1}} \{\beta_k\}$$

we have that

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

which proves the theorem. □

Note that, in order to apply Theorem 9, 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 non-preemptive mode. The algorithm for computing the longest non-preemptive intervals is illustrated in Figure 12.

```
Algorithm: Compute the Longest Non-Preemptive Intervals
Input: A task set $\mathcal{T}$ with $\{C_i, T_i, D_i, P_i\}, \forall \tau_i \in \mathcal{T}$
Output: $Q_i, \forall \tau_i \in \mathcal{T}$
// Assumes $\mathcal{T}$ is preemptively feasible and $D_i \leq T_i$
(1)  begin
(2)      $\beta_1 = D_1 - C_1$;
(3)      $Q_1 = \infty$;
(4)      for ($i := 2$ to $n$) do
(5)          $Q_i = \min\{Q_{i-1}, \beta_{i-1}\}$;
(6)          Compute $\beta_i$ using Equation (39) or (40);
(7)      end
(8)  end
```

Figure 12. Algorithm for computing the longest non-preemptive intervals.

6 Task splitting

According to this model, each task τ_i is split into m_i non-preemptive chunks (subjobs), obtained by inserting $m_i - 1$ preemption points in the code. Thus, preemptions can only occur at the subjobs boundaries. All the jobs generated by one task have the same subjob division. The k^{th} subjob has a worst-case execution time $q_{i,k}$, 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:

$$\begin{cases} q_i^{max} = \max_{k\in[1,m_i]} \{q_{i,k}\} \\ q_i^{last} = q_{i,m_i} \end{cases} \tag{43}$$

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:

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

For example, consider the same task set of Table 1, and suppose that τ_2 is split in two subjobs of 2 and 1 unit, and τ_3 is split in two subjobs of 4 and 2 units. The schedule produced by Deadline Monotonic with such a splitting is feasible and it is illustrated in Figure 13.

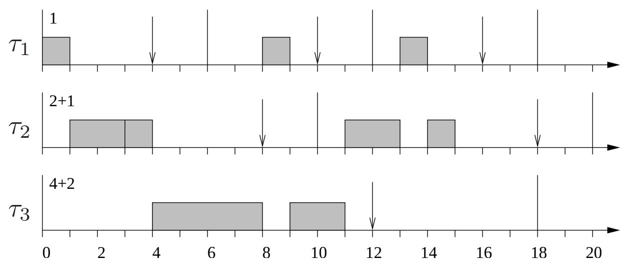


Figure 13. Schedule produced by Deadline Monotonic for the task set reported in Table 1, when τ_2 is split in two subjobs of 2 and 1 unit, and τ_3 is split in two subjobs of 4 and 2 units, respectively.

6.1 Feasibility Analysis

Feasibility analysis for task splitting can be carried out in a very similar way as the non preemptive 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:

$$B_i = \max_{j:P_j<P_i} \{q_j^{max} - 1\}. \tag{44}$$

- The last non-preemptive 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, which can be computed using the following recurrent relation:

$$\begin{cases} L_i^{(0)} = B_i + C_i \\ L_i^{(s)} = B_i + \displaystyle\sum_{h:P_h\geq P_i} \left\lceil \frac{L_i^{(s-1)}}{T_h} \right\rceil C_h. \end{cases} \tag{45}$$

In particular, L_i is the smallest value for which $L_i^{(s)} = L_i^{(s-1)}$. 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:

$$K_i = \left\lceil \frac{L_i}{T_i} \right\rceil. \tag{46}$$

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 and of the subjobs preceding the last one, and the interference of the tasks with priority higher than P_i. Hence,

$$\begin{aligned} s_{i,k} &= B_i + (k-1)C_i + (C_i - q_i^{last}) + \sum_{h:P_h>P_i} \left(\left\lfloor \frac{s_{i,k}}{T_h} \right\rfloor + 1 \right) C_h \\ &= B_i + kC_i - q_i^{last} + \sum_{h:P_h>P_i} \left(\left\lfloor \frac{s_{i,k}}{T_h} \right\rfloor + 1 \right) C_h. \end{aligned} \tag{47}$$

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

$$f_{i,k} = s_{i,k} + q_i^{last}. \tag{48}$$

Hence, the response time of task τ_i is given by

$$R_i = \max_{k \in [1,K_i]} \{f_{i,k} - (k-1)T_i\}. \tag{49}$$

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

$$\forall i = 1, \ldots, n \quad R_i \le D_i. \tag{50}$$

Assuming that the task set is preemptively feasible, the analysis can be simplified to the first job of each task, after the critical instant, as shown by Yao et al. [44]. Hence, the longest relative start time of τ_i can be computed as the smallest value satisfying the following recurrent relation:

$$S_i = B_i + C_i - q_i^{last} + \sum_{h:P_h > P_i} \left(\left\lfloor \frac{S_i}{T_h} \right\rfloor + 1 \right) C_h. \tag{51}$$

Then, the response time R_i is simply:

$$R_i = S_i + q_i^{last}. \tag{52}$$

6.2 Longest non-preemptive interval

As done in Section 5.2 under deferred preemptions, it is interesting to compute, also under task splitting, the longest non-preemptive interval Q_i for each task τ_i that can still preserve 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.

Tasks are assumed to be preemptively feasible, so that the analysis can be limited to the first job of each task. In this case, a bound on the blocking tolerance β_i can be achieved using the following schedulability condition [44]:

$$\exists t \in \mathcal{TS}^*(\tau_i) : B_i \le \{t - W_i^*(t)\}, \tag{53}$$

where $W_i^*(t)$ and the testing set $\mathcal{TS}^*$ are defined as

$$W_i^*(t) = C_i - q_i^{last} + \sum_{h:P_h > P_i} \left(\left\lfloor \frac{t}{T_h} \right\rfloor + 1 \right) C_h, \tag{54}$$

$$\mathcal{TS}^*(\tau_i) \doteq \mathcal{P}_{i-1}(D_i - q_i^{last}) \tag{55}$$

being $\mathcal{P}_i(t)$ given by Equation (9).

Rephrasing Equation (53), we obtain

$$B_i \le \max_{t \in \mathcal{TS}^*(\tau_i)} \{t - W_i^*(t)\}.$$

$$\beta_i = \max_{t \in \mathcal{TS}^*(\tau_i)} \{t - W_i^*(t)\}. \tag{56}$$

The longest non-preemptive interval Q_i that preserves feasibility for each task τ_i can then be computed by Theorem 9, using the blocking tolerances given by Equation (56). Applying the same substitutions, the algorithm in Figure 12 can also be used under task splitting.

As previously mentioned, the maximum length of the non-preemptive chunk under task splitting 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 $\beta_j(\tau_j \in hp(i))$, as expressed in Equation (41). Under task splitting, the blocking tolerance β_i is also a function of q_i^{last}, as clear form equations (54) and (56).

7 Selecting preemption points

When a task set is not schedulable in non-preemptive mode, there can be several ways to split tasks into subtasks to generate a feasible schedule, if there exists one. Moreover, as already observed in Section 1, the runtime overhead introduced by the preemption mechanism (including scheduling costs, cache and pipeline related delays) depends on the specific point where the preemption takes place. Hence, it would be useful to identify the best locations for placing preemption points inside each task to achieve a feasible schedule, while minimizing the overall preemption cost. This section illustrates an algorithm that achieves this goal.

7.1 Terminology and assumptions

Considering that there exist sections of code where preemption is not desirable (e.g., short loops, critical sections, I/O operations, etc.), each job of τ_i is assumed to consist of a sequence of N_i non-preemptive Basic Blocks (BBs), identified by the programmer based on the task structure. Preemption is allowed only at basic block boundaries, so each task has $N_i - 1$ *Potential Preemption Points* (PPPs), one between any two consecutive BBs. Critical sections and conditional branches are assumed to be executed entirely within a basic block. In this way, there is no need for using shared resource protocols to access critical sections.

The goal of the algorithm is to identify a subset of PPPs that minimizes the overall preemption cost, still preserving the schedulability of the task set. A PPP selected by the algorithm is referred to as an *Effective Preemption Point* (EPP), whereas the other PPPs are disabled. Therefore, the sequence of basic blocks between any two consecutive EPPs forms a Non-Preemptive Region (NPR). The following notation is used to describe the algorithm:

$\delta_{i,k}$ denotes the k-th basic block of task τ_i.

$b_{i,k}$ denotes the WCET of $\delta_{i,k}$ without preemption cost, that is, when τ_i is executed non-preemptively.

$\xi_{i,k}$ denotes the worst-case preemption overhead introduced when τ_i is preempted at the k-th PPP (i.e., between δ_k and δ_{k+1}).

N_i denotes the number of BBs of task τ_i, determined by the $N_i - 1$ PPPs defined by the programmer.

p_i denotes the number of NPRs of task τ_i, determined by the $p_i - 1$ EPPs selected by the algorithm.

$q_{i,j}$ denotes the WCET of the j-th NPR of τ_i, including the preemption cost.

$q_i^{\max}$ denotes the maximum NPR length for τ_i:

$$q_i^{\max} = \max\{q_{i,j}\}_{j=1}^{p_i}.$$

To simplify the notation, the task index is omitted from task parameters whenever the association with the related task is evident from the context. In the following, we implicitly refer to a generic task τ_i, with maximum allowed NPR length $Q_i = Q$. As shown in the previous sections, Q can be computed by the algorithm in Figure 12. We say that an EPP selection is *feasible* if the length of each resulting NPR, including the initial preemption overhead, does not exceed Q.

Figure 14 illustrates some of the defined parameters for a task with 6 basic blocks and 3 NPRs. PPPs are represented by dots between consecutive basic blocks: black dots are EPPs selected by the algorithm, while white dots are PPPs that are disabled. Above the task code, the figure also reports the preemption costs ξ_k for each PPP, although only the cost for the EPPs is accounted in the q_j of the corresponding NPR.

Using the notation introduced above, the non-preemptive WCET of τ_i can be expressed as follows:

$$C_i^{\text{NP}} = \sum_{k=1}^{N_i} b_{i,k}.$$

The goal of the algorithm is to minimize the overall worst-case execution time C_i of each task τ_i, including the preemption overhead, by properly selecting the EPPs among all the PPPs specified in the code by the programmer,

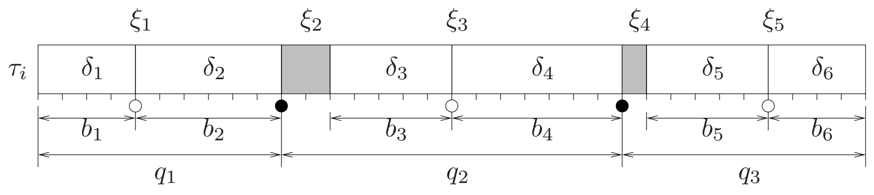


Figure 14. Example of task with 6 BBs split into 3 NPRs. Preemption cost is reported for each PPPs, but accounted only for the EPPs.

without compromising the schedulability of the task set. To compute the preemption overhead, we assume that each EPP leads to a preemption, and that the cache is invalidated after each context switch. Therefore,

$$C_i = C_i^{\text{NP}} + \sum_{k=1}^{N_i-1} \mathsf{selected}(i,k) \cdot \xi_{i,k}$$

where $\mathsf{selected}(i,k)$ = 1 if the k-th PPP of τ_i is selected by the algorithm to be an EPP, whereas $\mathsf{selected}(i,k)$ = 0, otherwise.

7.2 Proposed approach

First of all, it is worth noting that minimizing the number of EPPs does not necessarily minimize the overall preemption overhead. In fact, there are cases in which inserting more preemption points, than the minimum number, could be more convenient to take advantage of points with smaller cost.

Consider, for instance, the task illustrated in Figure 15, consisting of 6 basic blocks, whose total execution time in non preemptive mode is equal to $C_i^{NP} = 20$. The numbers above each PPP shown in Figure 15(a) denote the preemption cost, that is the overhead that would be added to C_i^{NP} if a preemption occurred in that location. Assuming a maximum non-preemptive interval $Q = 12$, a feasible schedule could be achieved by inserting a single preemption point at the end of δ_4, as shown in Figure 15(b). In fact, $\sum_{k=1}^{4} b_k = 3+3+3+2 = 11 \le Q$, and $\xi_4 + \sum_{k=5}^{6} b_k = 3+3+6 = 12 \le Q$, leading to a feasible schedule. This solution is characterized by a total preemption overhead of 3 units of time (shown by the gray execution area). However, selecting two EPPs, one after δ_1 and another after δ_5, a feasible solution is achieved with a smaller total overhead $\xi_1 + \xi_5 = 1 + 1 = 2$, as shown in Figure 15(c). In general, for tasks with a large number of basic blocks with different preemption cost, finding the optimal solution is not trivial.

For a generic task, the worst-case execution time q of a NPR composed of the consecutive basic blocks $\delta_j, \delta_{j+1}, \ldots, \delta_k$ can be expressed as

$$q = \xi_{j-1} + \sum_{\ell=j}^{k} b_\ell, \tag{57}$$

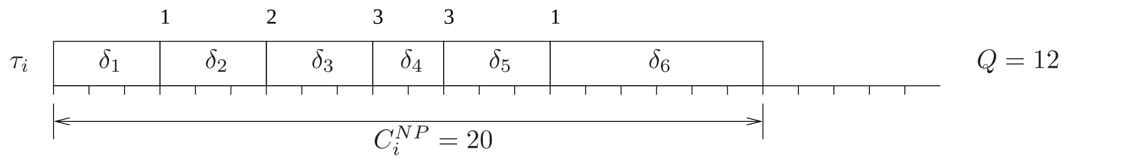
conventionally setting $\xi_0 = 0$. Note that the preemption overhead is included in q. Since any NPR of a feasible EPP selection has to meet the condition $q \le Q$, we must have

$$\xi_{j-1} + \sum_{\ell=j}^{k} b_\ell \le Q. \tag{58}$$

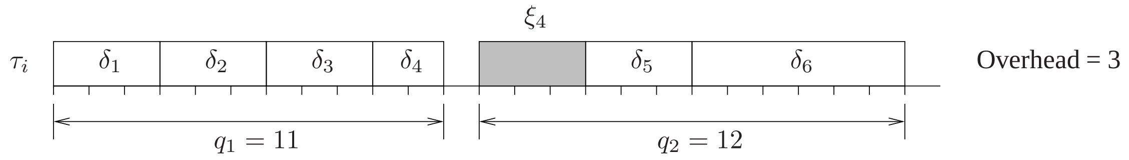
Now, let $\hat{C}_k$ be the WCET of the chunk of code composed of the first k basic blocks – i.e., from the beginning of δ_1 until the end of δ_k – including the preemption overhead of the EPPs that are contained in the considered chunk. Then, we can express the following recursive expression

$$\hat{C}_k = \hat{C}_{j-1} + q = \hat{C}_{j-1} + \xi_{j-1} + \sum_{\ell=j}^{k} b_\ell. \tag{59}$$

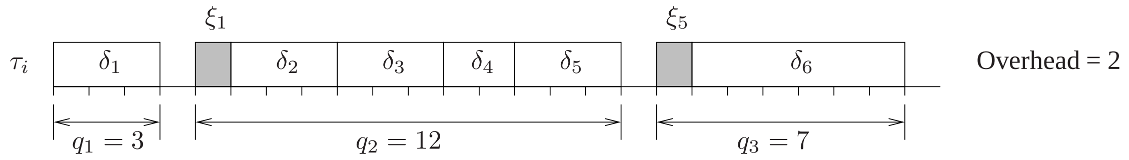
Note that since δ_N is the last BB, the worst-case execution time C_i of the whole task τ_i is equal to $\hat{C}_N$.



(a) Task with 6 basic blocks.



(b) Task with a single preemption point.



(c) Task with two preemption points.

Figure 15. Two solutions for selecting EPPs in a task with $Q = 12$: the first minimizes the number of EPPs, while the second minimizes the overall peremption cost.

The algorithm for the optimal selection of preemption points is based on the equations presented above and its pseudo-code is reported in Figure 16. The algorithm evaluates all the BBs in increasing order, starting from the first one. For each BB δ_k, the feasible EPP selection that leads to the smallest possible $\hat{C}_k$ is computed as follows.

For the first BBs, the minimum $\hat{C}_k$ is given by the sum of the BB lengths $\sum_{\ell=1}^{k} b_\ell$ as long as this sum does not exceed Q. Note that if $b_1 > Q$, there is no feasible PPP selection, and the algorithm fails. For the following BBs, $\hat{C}_k$ needs to consider the cost of one or more preemptions as well. Let $Prev_k$ be the set of the preceding BBs $\delta_{j \le k}$ that satisfy Condition (58), i.e., that might belong to the same NPR of δ_k. Again, if $\xi_{k-1} + b_k > Q$, there is no feasible PPP activation, and the algorithm fails. Otherwise, the minimum $\hat{C}_k$ is given by

$$\hat{C}_k = \min_{\delta_j \in Prev_k} \left\{ \hat{C}_{j-1} + \xi_{j-1} + \sum_{\ell=j}^{k} b_\ell \right\}. \tag{60}$$

Let $\delta^*(\delta_k)$ be the basic block for which the rightmost term of Expression (60) is minimum

$$\delta^*(\delta_k) = \min_{\delta_j \in Prev_k} \left\{ \hat{C}_{j-1} + \xi_{j-1} + \sum_{\ell=j}^{k} b_\ell \right\}. \tag{61}$$

If there are many possible BBs minimizing (60), the one with the smallest index is selected. Let $\delta_{Prev}(\delta_k)$ be the basic block preceding $\delta^*(\delta_k)$, if there exists one. The PPP at the end of $\delta_{Prev}(\delta_k)$ – or, equivalently, at the beginning of $\delta^*(\delta_k)$ – is meaningful for the analysis, since it represents the last PPP to activate for minimizing the preemption overhead of the first k basic blocks.

A feasible placement of EPPs for the whole task can then be derived with a recursive activation of PPPs, starting with the PPP at the end of $\delta_{Prev}(\delta_N)$, which will be the last EPP of the considered task. The penultimate EPP will be the one at the beginning of $\delta_{Prev}(\delta_{Prev}(\delta_N))$, and so on. If the result of this recursive lookup of function $\delta_{Prev}(k)$ is δ_1, the start of the program has been reached. A feasible placement of EPPs has therefore been detected, with a worst-case execution time, including preemption overhead, equal to $\hat{C}_N$. This is guaranteed to be the placement that minimizes the preemption overhead of the considered task, as proved in the next theorem.

Theorem 10. *The PPP activation pattern detected by procedure* ***SelectEPP***(τ_i, Q_i) *minimizes the preemption overhead experienced by a task* τ_i*, without compromising the schedulability.*

```
Algorithm: SelectEPP($\tau_i$, $Q_i$)
Input: $\{C_i, T_i, D_i, Q_i\}$ for task $\tau_i$
Output: The set of EPPs for task $\tau_i$
(1)  begin
(2)      $Prev_k := \{\delta_0\}$; $\hat{C}_0 := 0$;        // Initialize variables
(3)      for ($k := 1$ to $N$) do                 // For all PPPs
(4)          Remove from $Prev_k$ all $\delta_j$ violating (58);
(5)          if ($Prev_k = \emptyset$) then
(6)              return (INFEASIBLE);
(7)          end
(8)          Compute $\hat{C}_k$ using Equation (60);
(9)          Store $\delta_{Prev}(\delta_k)$;
(10)         $Prev_k := Prev_k \cup \{\delta_k\}$;
(11)     end
(12)     $\delta_j := \delta_{Prev}(\delta_N)$;
(13)     while ($\delta_j \neq \emptyset$) do
(14)         Select the PPP at the end of $\delta_{Prev}(\delta_j)$;
(15)         $\delta_j \leftarrow \delta_{Prev}(\delta_j)$;
(16)     end
(17)     return (FEASIBLE);
(18) end
```

Figure 16. Algorithm for selecting the optimal preemption points.

Proof. First, we prove that if procedure **SelectEPP**(τ_i, Q_i) fails, there is no other feasible EPP placement. For the procedure to fail, it is necessary that the condition at line (5) is satisfied for some δ_k. This means that δ_k violates Condition (58) for any $j \leq k$. That is,

$$\xi_{j-1} + \sum_{\ell=j}^{k} b_\ell > Q, \qquad \forall j \leq k.$$

This means that with any possible PPP activation pattern, the length of the NPR containing δ_k will be larger than Q, leading to a deadline miss.

We now consider the minimization of the preemption overhead. Let C_i be the WCET, including the preemption overhead, resulting from the EPP allocation given by **SelectEPP**(τ_i, Q_i). Suppose there exists another feasible EPP allocation that results in a smaller $C_i' < C_i$. Then, there should be a BB $\delta_k, 1 \leq k \leq N$ for which $\hat{C}_k' < \hat{C}_k$. We prove by induction that this is not possible. The proof inducts over the index j of the basic blocks δ_j, proving that $\hat{C}_j$ is minimized for all $j, 1 \leq j \leq k$.

Base case For $j = 1$, $\hat{C}_1 = b_1$ by definition. This is the minimum possible value of the WCET of the first BB, since it does not experience any preemption.

Inductive step Assume all $\hat{C}_\ell$, $\forall \ell < j$ are minimized by procedure **SelectEPP**(τ_i, Q_i). We prove that $\hat{C}_j$ is also minimized. By Equation (60), procedure **SelectEPP**(τ_i, Q_i) computes $\hat{C}_j$ as

$$\hat{C}_j = \min_{\delta_\ell \in Prev_j} \left\{ \hat{C}_{\ell-1} + \xi_{\ell-1} + \sum_{m=\ell}^{j} b_m \right\}.$$

Since, by induction hypothesis, all $\hat{C}_{\ell-1}$ terms are minimal, also $\hat{C}_j$ is minimized, proving the statement. Therefore, $\hat{C}_k$ is the minimum possible value, reaching a contradiction and proving the theorem. □

The feasibility of a given task set is maximized by applying procedure **SelectEPP**(τ_i, Q_i) to each task τ_i, starting from τ_1 and proceeding in task order. Once the optimal allocation of EPPs has been computed for a task τ_i, the value of the overall WCET $C_i = \hat{C}_N$ can be used for the computation of the maximum allowed NPR Q_{i+1} of the next task τ_{i+1}, using the technique presented in Section 6. The procedure is repeated until a feasible PPP activation pattern has been produced for all tasks in the considered set. If the computed Q_{i+1} is too small to find a feasible EPP allocation, the only possibility to reach schedulability is by removing tasks from the system, as no other EPP allocation strategy would produce a feasible schedule.

8 Assessment of the approaches

The limited preemption methods presented here can be compared under several aspects, such as:

- Implementation complexity.
- Predictability in estimating the preemption cost;
- Effectiveness in reducing the number of preemptions;

8.1 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.

In deferred preemption (floating model), non-preemptive 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 time-triggered mode, non-preemptive regions can be realized by setting a timer to enforce the maximum interval in which preemption is disabled. Initially, all tasks can start executing in non-preemptive 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 non-preemptive 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.

8.2 Predictability

As observed in Section 1, 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, the location where the running task is preempted is set q_i units of time ahead of the arrival time of a higher priority task. Hence, it depends on the actual execution of the running task and on the arrival time of higher priority task. Therefore, it can hardly be predicted off line.

On the contrary, under cooperative scheduling, the locations where preemptions occur are defined by the programmer at design time, hence the corresponding overhead can be estimated more precisely by timing analysis tools. Moreover, through the algorithm presented in Section 7.2, it is also possible to select the best locations for placing the preemption points to minimize the overall preemption cost.

8.3 Effectiveness

Each of the presented methods can be used to limit preemption as much as desired, but 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, un 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:

$$\nu_i = \sum_{h:P_h>\theta_i} \left\lceil \frac{R_i}{T_h} \right\rceil .$$

This is an upper bound because simultaneous activations are counted as they were different, although they cause a single preemption.

Under deferred preemption, the number of preemptions occurring on τ_i is easier to determine, because it directly depends on the non-preemptive interval q_i specified for the task. If preemption cost is neglected, we simply have:

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

However, if preemption cost is not negligible, the estimation requires an iterative approach, since the task computation time also depends on the number of preemptions. Considering a fixed cost ξ_i for each preemption, then the number of preemptions can be upper bounded using the following recurrent relation:

$$\begin{cases} \nu_i^0 = \left\lceil \frac{C_i^{NP}}{q_i} \right\rceil - 1 \\ \\ \nu_i^s = \left\lceil \frac{C_i^{NP}+\xi_i \nu_i^{s-1}}{q_i} \right\rceil - 1 \end{cases}$$

where the iteration process converges when $\nu_i^s = \nu_i^{s-1}$.

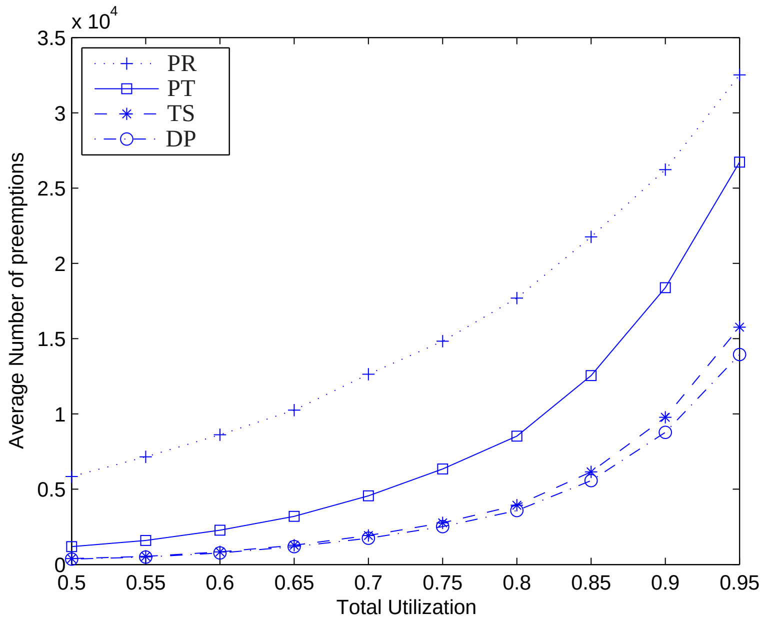
Finally, under cooperative scheduling, the number of preemptions can be simply upper bounded by the number of effective preemption points inserted in the task code.

A set of simulations with randomly generated task sets have been carried out in [46] to better evaluate the effectiveness of the considered algorithms in reducing the number of preemptions. Each simulation run was performed on a set of n tasks with total utilization U varying from 0.5 to 0.95 with step 0.05. Individual utilizations U_i were uniformly distributed in [0,1], using the UUniFast algorithm [12]. Each computation time C_i was generated as a random integer uniformly distributed in [10, 50], and then T_i was computed as $T_i = C_i/U_i$. The relative deadline D_i was generated as a random integer in the range $[C_i + 0.8 \cdot (T_i - C_i),\ T_i]$. The total simulation time was set to 1 million units of time. For each point in the graph, the result was computed by taking the average over 1000 runs.

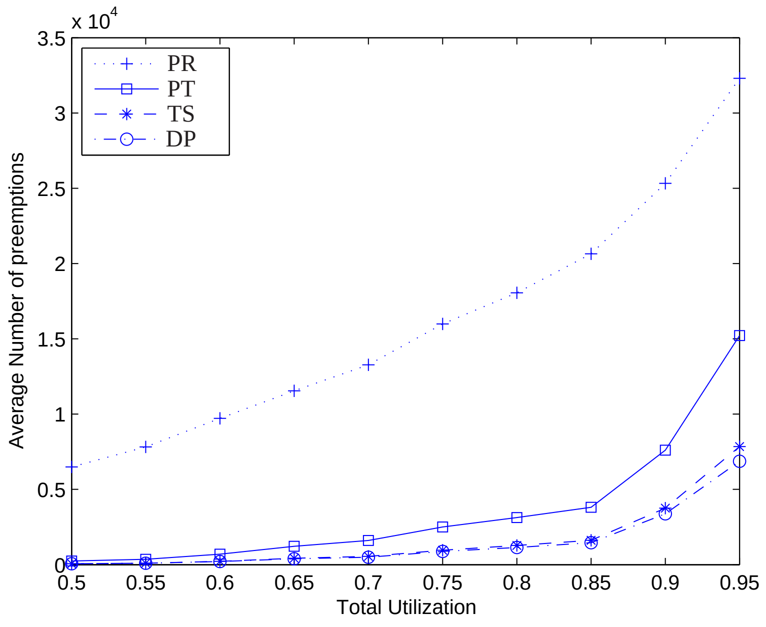
Figures 17(a) and 17(a) show the simulation results obtained for a task set of 6 and 12 tasks, respectively, and report the number of preemptions produced by each method as a function of the load.

All the task sets have been generated to be preemptive feasible and the preemption cost was ignored, as also done in [43, 45]. Under preemption thresholds (PT), the algorithm proposed by Saksena and Wang [38] was used to find the maximum priority threshold that minimize the number of preemptions. Under deferred preemptions (DP) and task splitting (TS), the longest non-preemptive regions were computed according to the methods presented in Sections 5.2 and 6.2, respectively. Finally, under task splitting, preemption points were inserted from the end of task code to the beginning.

As expected, fully preemptive scheduling (PR) generates the largest number of preemptions, while DP and TS are both able to achieve a higher reduction. PT has an intermediate behavior. Notice that DP can reduce slightly more preemptions than TS since, on the average, each preemption is deferred for a longer interval (always equal to Q_i, except when the preemption occur near the end of the task). However, it is important to consider that TS can achieve a lower and more predictable preemption cost, since preemption points can be suitably decided off line with this purpose. As showed in the figures, PR produces a similar number of preemptions when the number of tasks increases, whereas all the other methods reduce the number of preemptions to a even higher degree. This is because, when n is larger, tasks have smaller individual utilization, thus can suffer more blocking from lower priority tasks.



(a) Number of tasks: n=6



(b) Number of tasks: n=12

Figure 17. Average number of preemptions with different number of tasks.

9 Conclusions

This work was motivated by the observation that preemptive scheduling degrades the predictability of an application, making WCETs significantly higher and more difficult to estimate. Although, disabling preemptions would be very effective for solving the problem, in most cases non-preemptive scheduling would introduce large blocking times in high-priority tasks that would jeopardize the schedulability. Therefore, this work presented and analyzed three scheduling methods for limiting preemptions during task execution, while preserving the schedulability of the task set.

The results achieved in this work can be summarized in Table 3, which compares the three presented methods in terms of the metrics presented above. As discussed in the previous section, the preemption threshold mechanism can reduce the overall number of preemptions with a low runtime overhead, however preemption cost cannot be easily estimated, since the position of each preemption and the overall number of preemptions for each task cannot be determined off line. Using deferred preemptions, the number of preemptions for each task can be better estimated, but still the position of each preemption cannot be determined off line. Cooperative Scheduling is the most predictable mechanism for estimating preemption costs, since both the number of preemptions and their positions are fixed and known from the task code. Its implementation, however, requires inserting explicit system calls in the source code that introduce additional overhead.

	Implementation cost	Predictability	Effectiveness
Preemption Thresholds	Low	Low	Medium
Deferred Preemptions	Medium	Medium	High
Cooperative Scheduling	Medium	High	High

Table 3. Evaluation of limited preemption methods.

References

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

[2] N.C. Audsley, A. Burns, M.F. Richardson, K. Tindell, and A.J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.

[3] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *IEEE Workshop on Real-Time Operating Systems*, 1992.

[4] T.P. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3, 1991.

[5] Sanjoy Baruah. Resource sharing in edf-scheduled systems: a closer look. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS'06)*, Rio de Janeiro, Brazil, December 5-8, 2006.

[6] Sanjoy Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *Proc. of the 17th Euromicro Conf. on Real-Time Systems (ECRTS'05)*, pages 137–144, Palma de Mallorca, Balearic Islands, Spain, July 6-8, 2005.

[7] S.K. Baruah, L.E. Rosier, and R.R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Journal of Real-Time Systems*, 2, 1990.

[8] Marko Bertogna, Giorgio Buttazzo, Mauro Marinoni, Gang Yao, Francesco Esposito, and Marco Caccamo. Preemption points placement for sporadic task sets. In *Proc. of the 22th Euromicro Conf. on Real-Time Systems (ECRTS'10)*, Brussels, Belgium, July 6-9, 2010.

[9] Enrico Bini, Giorgio Buttazzo, and Giuseppe Buttazzo. A hyperbolic bound for the rate monotonic algorithm. In *Proceedings of the IEEE Euromicro Conference on Real-Time Systems*, pages 59–66, June 2001.

[10] Enrico Bini, Giorgio Buttazzo, and Giuseppe Buttazzo. Rate monotonic scheduling: The hyperbolic bound. *IEEE Transactions on Computers*, 52(7):933–942, July 2003.

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

[12] Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[13] 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. *Real-Time System*, 42(1-3):63–119, 2009.

[14] Bach D. Bui, Marco Caccamo, Lui Sha, and Joseph Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. In *IEEE Proceedings of the 14th Int. Conf. on Embedded and Real-Time Computing Systems and Applications*, pages 101–110, Kaohsiung, Taiwan, August 2008.

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

[16] J.V. Busquets-Matraix. Adding instruction cache effect to an exact schedulability analysis of preemptive real-time system. In *Proc. of the 8th Euromicro Workshop on Real-Time Systems*, L'Aquila, Italy, June 12-14, 1996.

[17] Giorgio Buttazzo and Anton Cervin. Comparative assessment and evaluation of jitter control methods. In *Proceedings of the 15th Int. Conf. on Real-Time and Network Systems (RTNS'07)*, pages 137–144, Nancy, France, March 29-30, 2007.

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

[19] J. Echague, I. Ripoll, and A. Crespo. Hard real-time preemptively scheduling with high context switch cost. In *Proc. of the 7th Euromicro Workshop on Real-Time Systems*, Odense, Denmark, June 14-16, 1995.

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

[21] Paolo Gai, Luca Abeni, Massimiliano Giorgi, and Giorgio Buttazzo. A new kernel approach for modular real-time systems development. In *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*, Delft, The Netherlands, June 2001.

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

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

[24] Laurent George, Nicolas Rivierre, and Marco Spuri. Preemptive and non-preemptive real-time uniprocessor scheduling. Research Report RR-2966, INRIA, France, 1996.

[25] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1998.

[26] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks with varying execution priority. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 129–139, December 1991.

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

[28] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

[29] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the 10th IEEE Real-Time Systems Symposium (RTSS'89)*, pages 166–171, Santa Monica, CA, USA, December 5-7, 1989.

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

[31] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Proc. of ACM Workshop on Experimental Computer Science (ExpCS'07)*, San Diego, California, June 13-14, 2007.

[32] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1), 1973.

[33] A.K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA, 1983.

[34] Harini Ramaprasad and Frank Mueller. Tightening the bounds on feasible preemptions. *ACM Transactions on Embedded Computing Systems*, 2009. To appear.

[35] Harini Ramaprasad and Frank Mueller. Bounding worst-case response time for tasks with non-preemptive regions. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS'08)*, pages 58–67, St. Louis, MO, USA, April 22-24, 2008.

[36] Harini Ramaprasad and Frank Mueller. Tightening the bounds on feasible preemption points. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS'06)*, pages 212–224, Rio de Janeiro, Brazil, December 5-8, 2006.

[37] John Regehr. Scheduling tasks with mixed preemption relations for robustness to timing faults. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, pages 315–326, Austin, TX, USA, December 3-5, 2002.

[38] Manas Saksena and Yun Wang. Scalable real-time system design using preemption thresholds. In *Proc. of the 21st IEEE Real-Time Systems Symposium (RTSS'00)*, Orlando, Florida, USA, November 27-30, 2000.

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

[40] John A. Stankovic and Krithi Ramamritham. The spring kernel: A new paradigm for real-time systems. *IEEE Software*, 8(3):62–72, May 1991.

[41] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Proc. of the 17th Euromicro Conf. on Real-Time Systems (ECRTS'05)*, pages 41–48, Palma de Mallorca, Balearic Islands, Spain, July 6-8, 2005.

[42] Jan Staschulat and Rolf Ernst. Multiple process execution in cache related preemption delay analysis. In *Proc. of the 4th ACM Int. Conf. on Embedded Software (EMSOFT'04)*, pages 278–286, Pisa, Italy, September 27-29, 2004.

[43] Yun Wang and Manas Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Proc. of the 6th IEEE Int. Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, pages 328–335, Hong Kong, China, December 13-15, 1999.

[44] Gang Yao, Giorgio Buttazzo, and Marko Bertogna. Feasibility analysis under fixed priority scheduling with fixed preemption points. In *Proc. of the 16th IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA'10)*, pages 71–80, Macau, SAR, China, August 23-25, 2010.

[45] Gang Yao, Giorgio Buttazzo, and Marko Bertogna. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. In *Proc. of the 15th IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA'09)*, pages 351–360, Beijing, China, August 24-26, 2009.

[46] Gang Yao, Giorgio Buttazzo, and Marko Bertogna. Comparative evaluation of limited preemptive methods. In *Proc. of the 15th IEEE Int. Conf. on Emerging Techonologies and Factory Automation (ETFA 2010)*, Bilbao, Spain, September 13-16, 2010.

[47] Patrick Meumeu Yomsi and Yves Sorel. Extending rate monotonic analysis with exact cost of preemptions for hard real-time systems. In *Proc. of the 19th EuroMicro Conf. on Real-Time Systems (ECRTS'07)*, Pisa, Italy, July 4-6, 2007.