Proc. of the 22nd Euromicro Conference on Real-Time Systems (ECRTS 10), Brussels, Belgium, July 6-9, 2010.

Preemption points placement for sporadic task sets

Marko Bertogna, Giorgio Buttazzo, Mauro Marinoni,
Gang Yao, Francesco Esposito
Scuola Superiore Sant'Anna
Pisa, Italy
Email: {name.surname}@sssup.it

Marco Caccamo
University of Illinois at Urbana Champaign
Urbana, IL
Email: mcaccamo@illinois.edu

***Abstract*—Limited preemption scheduling has been introduced as a viable alternative to non-preemptive and fully-preemptive scheduling when reduced blocking times need to coexist with an acceptable context switch overhead. To achieve this goal, preemptions are allowed only at selected points of the code of each task, decreasing the preemption overhead and simplifying the estimation of worst-case execution parameters. Unfortunately, the problem of how to place these preemption points is rather complex and has not been solved.**

In this paper, a method is presented for the optimal placement of preemption points under simplifying conditions, namely, a fixed preemption overhead at each point. We will prove that if our method is not able to produce a feasible schedule, then no other possible preemption point placement (including non-preemptive and fully preemptive scheduling) can find a schedulable solution. The presented method is general enough to be applicable to both EDF and Fixed Priority scheduling, with limited modifications.

I. Introduction

In safety-critical applications, the use of advanced real-time scheduling techniques is significantly limited by the difficulty of finding tight estimations of worst-case execution parameters. To simplify the problem, most theoretical results on schedulability analysis have been derived assuming a preemption cost equal to zero. Under such an ideal case, preemptive scheduling is often more efficient than non-preemptive scheduling, because of the additional blocking time that can be introduced by the non-preemptive execution of lower priority tasks. In practice, however, preemptions can introduce a significant runtime overhead and may cause high fluctuations in task execution times, therefore degrading system predictability. In particular, the following types of costs must be taken into account at each preemption:

1) *Scheduler cost*. It is due to the time taken by the scheduling algorithm to suspend the running task, insert it into the ready queue, switch the context, and dispatch the new incoming task.
2) *Pipeline cost*. It is due to the time taken to flush the processor pipeline when the task is interrupted and the time taken to refill the pipeline when the task is resumed.
3) *Cache-related cost*. It is due to the time taken to reload the cache lines evicted by the preempting task. This time depends on the specific point in which preemption occurs and on the number of preemptions experienced by the task [18], [12], [1].
4) *Bus contention cost*. It is due to the Front Side Bus (FSB) conflicts caused by the extra memory accesses due to cache misses. In fact, whenever data are not found in the cache, they have to be fetched from RAM, using the FSB. Hence, contentions can occur when the FSB is used by I/O peripheral devices through a DMA transfer [21], [20].

These effects are not negligible at all, and may contribute to a great share of the overall worst-case execution time (WCET). To overcome such problems, some authors investigated limited preemption models that can be used to reduce the negative effects of context switches, while limiting the amount of blocking due to non preemptive regions [29], [26], [8], [2], [32]. From another side, other authors extended the schedulability analysis of preemptive scheduling to take context switch overhead into account [11], [33]. The problem of selecting preemption points in order to improve the schedulability of the system has been preliminarily considered in [13] and [17].

Indeed, such a problem is not easy to solve in an optimal way, since it is characterized by a circular dependency. In fact, when considering the context switch overhead in the schedulability analysis, the WCET of a task becomes a function of the number of preemptions it might be subject to; but the number of preemptions depends on its turn by the WCET of the task—the longer a task executes, the more it will be preempted—complicating the analysis.

In this paper we will show how to deal with such circular dependency, when a limited preemption model with fixed preemption points is adopted. The advantage of this model is that it is in line with the current practice adopted in critical software development [8], so that the derived results can be applied to real applications. We will present a method for automatically selecting the most suitable preemption points in the code of each task in order to guarantee the schedulability of the system. The analysis will consider on one hand the increased blocking caused by non-preemptive sections, and on the other hand the beneficial reduction of the preemption overhead. We will prove that the proposed algorithm is optimal when each preemption point is assumed to produce an identical overhead. Even if this assumption could appear rather restrictive, we introduced it to establish the mathematical background for more complex models

without violating the strict page limit of this submission. As preliminarily shown in [31], the presented analysis can be integrated with data collected by timing analysis tools, for the handling of more realistic models with a variable preemption overhead.

The proposed approach is general enough to be applicable to the most used scheduling algorithms, such as EDF and FP. We will show how existing results on limited preemption scheduling can be extended and integrated under a common notational model, in order to derive the necessary information for the optimal placement of preemption points. To comply with more general requirements, we will also analyze the case in which preemption points can be inserted only at a discrete number of points. This will allow our algorithm to deal with user-defined non-interruptible sections of code, as well as avoid complex protocols for access to shared resources. In fact, when it is possible to encapsulate each critical section within a non-preemptive region, a task will never be preempted while holding a lock, solving any mutual exclusion problem in the access to shared resources.

II. Related Work

A. Non-Preemptive and Limited Preemption scheduling

Non-preemptive EDF scheduling has been studied by Jeffay et al. [14], who showed that EDF is optimal even among non-preemptive work-conserving schedulers[1] for periodic and sporadic task sets. For these systems, an exact schedulability test with pseudo-polynomial complexity was provided. Moreover, it was shown that, for concrete periodic task systems scheduled by non-preemptive algorithms[2], feasibility analysis is NP-hard in the strong sense.

Baruah and Chakraborty [3] analyzed the schedulability of non-preemptive task sets under the recurring task model, deriving polynomial time approximation algorithms for both preemptive and non-preemptive scheduling.

Wang and Saksena [29] proposed a different approach for limiting preemptions, in systems scheduled with FP. Each task is assigned a regular priority and a preemption threshold, and it is allowed to preempt only when its priority is higher than the threshold of the preempted task. This work has been later improved by Regehr in [26].

Burns [8] extended the response time analysis to verify the schedulability of fixed priority tasks with fixed preemption points. His work has been later improved by Bril et al. [7].

Baruah introduced limited preemption scheduling for EDF [2], computing the maximum amount of time for which a task may execute non preemptively without missing any deadline. Yao *et al.* [32] extended Baruah's work to fixed priority systems.

B. Preemption overhead

The problem of finding a correct WCET estimation for real-time task sets has been considered in many different papers in the timing analysis domain (see [30] for a good survey). When a preemptive scheduler is adopted, a critical factor in the estimation of a task's WCET is represented by the Cache-Related Preemption Delay (CRPD).

In [9] and [22], two methods have been presented to integrate the classic Response Time Analysis with the penalties associated with CRPD, adding a fixed context-switch cost. A complex but more precise analysis considering common sets of data between preempting and preempted tasks has been described in [16]. With a similar target, Staschulat *et al.* [28] provided safe estimations of the CRPD, analyzing the intersection between the set of *useful* data—locations that might be accessed again by a preempted task—and *used* data—locations that might be accessed by the preempting task. The appropriate selection of preemption points for an easier computation of the CRPD has been addressed in [27].

In [23], a bound was provided on the Data Cache Related Preemption Delay (D-CRPD), identifying additional data-cache misses due to context switches. Response Time Analysis was then used to check the system schedulability, using the derived bound on the worst-case execution times. This bound was then refined in [24]. In a recent work [25], the same authors extended the analysis to tasks having at most one non-preemptive region with a given position inside the task code.

While most of the above works were based on systems scheduled with Fixed Priority, Ju *et al.* [15] considered the CRPD computation problem for systems scheduled with preemptive EDF.

C. Improvements over previous works

In this paper, we consider the problem of scheduling a set of real-time tasks consisting of a sequence of Non-Preemptive Regions (NPR) separated by Preemption Points (PP). The proposed method helps a designer in selecting the best preemption points, exploiting the available slack in the system to reduce the number of preemptions of some selected tasks, without imposing too much blocking on higher priority tasks. The final objective is to achieve a feasible schedule when the task set is not feasible in non-preemptive mode (due to high blocking times), nor in fully preemptive mode (due to the high overhead).

As shown in [32], [4], limited preemption schedulers can significantly reduce the total number of preemptions with respect to fully preemptive algorithms. This happens because the allowed non-preemptive execution length of a task is often larger than or comparable to that task's execution time. However, existing theoretical results on limited preemption scheduling [2], [4], [32] have been derived neglecting the cost of preemptions. Integrating these results with the preemption overhead is not so straightforward, since computing the maximum lengths of the non-preemptive regions requires the knowledge of worst-case execution times, which in turn are significantly influenced by the number of context switches. In this paper, we show how to deal with such a circular dependency, proposing an iterative algorithm that considers both problems at the same time. Earlier attempts

[1] A scheduling algorithm is work-conserving if the processor is never idled when a task is ready to execute. Note that EDF is not optimal among general non-preemptive schedulers (including non work-conserving ones).

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

to reduce context switching overhead delaying preemptions have been presented in [13] and [17].

The rest of the paper is organized as follows. In Section III, we will present the adopted system model and terminology. Section IV describes a schedulability analysis for task sets scheduled with limited preemption EDF or FP. In Section V, we will show an algorithm to achieve the schedulability of a task set with a proper placement of preemption points inside each task's code. In Section VI, we will present some considerations on the proposed method. The effectiveness of this method will be evaluated through a set of simulations, shown in Section VII. Finally, we will draw our conclusions in Section VIII.

III. System Model

We consider a set τ of n periodic and sporadic real-time tasks that are scheduled on a single processor using either a fixed priority algorithm (FP) or Earliest Deadline First (EDF) [19]. Each task τ_i is defined by a worst-case execution requirement C_i, a period, or minimum interarrival time, T_i, and a relative deadline $D_i \leq T_i$. Each task generates an infinite sequence of jobs, with the first job arriving at any time and successive job-arrivals separated by at least T_i time units. The utilization U_i of task τ_i is defined as C_i/T_i. The total utilization U of task set τ is the sum of the utilizations of all tasks in τ. We assume that tasks are ordered by decreasing priorities in the FP case, and by increasing relative deadlines in the EDF case, i.e., $\forall i \mid 0 < i < n : D_{i-1} \leq D_i$. Tasks are either supposed to be independent, or their critical sections are assumed to be entirely contained within a non-preemptive region[3].

Each job of τ_i consists of a sequence of p_i non-preemptive chunks of code. Preemption is allowed only between chunks by inserting proper preemption points. The j-th chunk of task τ_i is denoted by $\delta_{i,j}, 1 \leq i \leq n, 1 \leq j \leq p_i$, and its worst-case execution time by $q_{i,j}$. The maximum chunk length for τ_i is $q_i^{\max} = \max\{q_{i,j}\}_{j=1}^{p_i}$.

The *memory footprint* F_i of a task τ_i is the cumulative size of the individual memory locations accessed by a job of τ_i during its execution. A task repeatedly accessing the same set of data will have a smaller footprint than a task accessing multiple different memory locations.

We assume the processor can take advantage of a dedicated cache, of size L, from which recently used data and instructions can be loaded. We say that a cache is "cold" if it does not contain any useful data; otherwise, the cache is "hot". A cache that is always hot is referred to as an "ideal cache". The cache miss penalty due to the time taken to load data from the main memory to the cache is denoted by γ. To simplify the analysis, we assume this value to be the same for every memory location accessed by each task in the set. Moreover, we ignore any timing anomaly in the cache behavior, assuming each miss increases the observed execution time by γ. Finally, we assume the cache being completely *cold* after any context switch. In other words, we do not take advantage of the positive cache effects due to the subsequent execution of concurrent tasks accessing similar sets of data, nor to the limited number of cache evictions performed by a preempting job with reduced footprint (smaller than the cache size).

These restricting assumptions will be removed in a future work, where less pessimistic estimations of the CRPD will be considered[4]. Before complicating the model, this paper intends to presents the preliminary results that are needed for a more thorough analysis.

A. Worst-case execution times

The worst-case execution time C_i of a task τ_i is the largest amount of processor time a job of τ_i might need to successfully complete its execution. To perform a precise schedulability analysis, this parameter must include all overhead costs identified in the introduction, and can be expressed as the sum of the net computation time E_i, (achieved when all accessed data are always in the cache) plus such penalties.

In particular, the maximum number of cache misses a task τ_i may experience in the worst-case scenario is denoted by $\mu_i^{\max}$, and it is equal to the maximum number of memory accesses a job of τ_i may perform. Indeed, this is the only bound that can be given when no information is available on the adopted scheduler, nor on the tasks concurrently scheduled with τ_i.

When a particular scheduler is assumed, the estimation of the real number of cache misses may be refined. We call μ_i the maximum number of cache misses a task τ_i may experience using a given scheduling algorithm. For instance, with preemptive EDF or FP it has been shown [10] that the number of preemptions on a job of task τ_i is bounded by the number of higher priority jobs that can be released in $[0, D_i)$[5], decreasing the number of potential cache misses in the worst-case. When τ_i is executed non-preemptively, μ_i has the smallest possible value μ_i^{NP}. Hence, the following relation holds:

$$\mu_i^{\text{NP}} \leq \mu_i \leq \mu_i^{\max}.$$

Note that μ_i depends on the number p_i of non-preemptive regions in which τ_i is divided. The smaller p_i, the fewer the cache misses experienced by τ_i. In fact, each context switch might evict the cache locations commonly accessed by two subsequent chunks. To understand that, consider the example shown in Figure 1, where the memory accesses of the first two chunks of a task τ_i are shown. The first chunk loads into the cache the memory locations corresponding to `a`, `b` and `c`. When the second chunk starts executing after a potential preemption, another task might have overwritten the cache content, evicting data commonly accessed by $\delta_{i,1}$ and $\delta_{i,2}$. Therefore, $\delta_{i,2}$'s first accesses to `a` and `c` should be accounted as misses. To clarify which misses are due to a possible preemption and which are not, we distinguish

[3] As critical sections are typically very short [6], they are likely to be accommodated inside a non-preemptive region. When this is not true, some shared resource protocol needs to be adopted.

[4] Some insights of this future work can be found in [31]

[5] See [11], [28], [24], [33] for tighter bounds in the number of preemptions.

Chunk	Code	Hit/Miss
$\delta_{i,1}$	`access(a)`	I
	`access(b)`	I
	`access(c)`	I
	`access(a)`	H
	`access(c)`	H
$\delta_{i,2}$	`access(a)`	E
	`access(d)`	I
	`access(a)`	H
	`access(c)`	E
	`access(d)`	H
	...	

Figure 1. Example of cache accesses: (H) cache Hit, (I) Intrinsic miss, (E) Extrinsic miss.

Symbol	Description
$\delta_{i,j}$	j-th chunk of task τ_i
p_i	Number of chunks of task τ_i
C_i	WCET of τ_i in presence of cache misses
C_i^{NP}	WCET of τ_i when it executes non-preemptively
E_i	WCET value with an ideal cache
$q_{i,j}$	WCET of chunk $\delta_{i,j}$
$q_i^{\max}$	Largest non-preemptive execution of τ_i
μ_i	Worst-case number of cache misses of τ_i
$\mu_i^{\max}$	Maximum μ_i among all possible schedulers
μ_i^{NP}	μ_i value when τ_i executes non-preemptively
L	Cache size
F_i	Memory footprint of task τ_i
γ	Cache miss penalty
σ	Penalty due to load/store the task state
π	Penalty due to pipeline invalidation
$\eta(x)$	I/O induced delay for x cache misses

Figure 2. Notation used throughout the paper.

between *intrinsic* and *extrinsic* cache misses. A miss is intrinsic if it occurs independently of the preemption, i.e., when a task accesses a memory location for the first time, or when the miss is caused by a self-eviction[6]. An extrinsic miss is instead due to evictions caused by preempting tasks.

As already mentioned in the introduction, there are also other kinds of penalties associated to each preemption, like the scheduler cost σ, the pipeline cost π, and the FSB contention cost $\eta(\mu_i)$. Since there are p_i non-preemptive chunks, the total number of times a job of τ_i may be preempted is $(p_i - 1)$. Hence, the overall worst-case execution time of τ_i results to be

$$C_i = E_i + \gamma\mu_i + (\pi + \sigma)(p_i - 1) + \eta(\mu_i). \quad (1)$$

In the next sections, we will present a method to decrease this value by minimizing the number p_i of preemption points inside the code of each task τ_i, resulting in a smaller number of cache misses μ_i.

For convenience, all notations are summarized in Figure 2.

IV. Schedulability analysis

In this section, we present a unified analysis of EDF and FP scheduling under the limited preemption model, extending and reformulating the results derived in [32] (for FP) and in [2], [4] (for EDF), under a common notational model.

For the feasibility analysis under FP, we use the *request bound function* $\text{RBF}_i(a)$ in an interval a, defined as

$$\text{RBF}_i(a) = \left\lceil \frac{a}{T_i} \right\rceil C_i.$$

Under EDF, the analysis is carried out by the *demand bound function* $\text{DBF}_i(a)$ in an interval a, defined as

$$\text{DBF}_i(a) = \left(1 + \left\lfloor \frac{a - D_i}{T_i} \right\rfloor\right) C_i.$$

Moreover, we conventionally set D_{n+1} equal to the minimum between: (i) the least common multiple (lcm) of $T_1, T_2, \ldots, T_n$, and (ii) the following expression[7]:

$$\max\left(D_n, \frac{1}{1-U} \cdot \sum_{i=1}^{n} U_i \cdot \max\left(0, T_i - D_i\right)\right).$$

The largest blocking B_i that a task τ_i might experience is given, under both FP and EDF, by the length of the largest non-preemptive chunk belonging to tasks with index higher than i:

$$B_i = \max_{i<k\leq n+1}\{q_k^{\max}\}, \quad (2)$$

where $q_{n+1}^{\max} = 0$ by definition. Summarizing the results presented in [32], [2], [4], the next theorem derives a schedulability condition under limited preemptions, for FP and EDF.

Theorem 1. *A task set τ is schedulable with limited preemption* EDF *or* FP *if, for all $i \mid 1 \leq i \leq n$,*

$$B_i \leq \beta_i, \quad (3)$$

where, under FP, *β_i is given by*

$$\beta_i^{\text{FP}} \doteq \max_{a\in A | a\leq D_i}\left\{a - \sum_{j\leq i}\text{RBF}_j(a)\right\}, \quad (4)$$

with

$$A = \{kT_j,\ k \in \mathbb{N},\ 1 \leq j < n\},$$

whereas, under EDF, *β_i is given by*

$$\beta_i^{\text{EDF}} \doteq \min_{a\in A | D_i\leq a< D_{i+1}}\left\{a - \sum_{\tau_j\in\tau}\text{DBF}_j(a)\right\}, \quad (5)$$

with

$$A = \{kT_j + D_j,\ k \in \mathbb{N},\ 1 \leq j \leq n\}.$$

The following theorem presents a different schedulability condition, expressed in terms of a bound Q_k on the longest non-preemptive region $q_k^{\max}$ of each task τ_k.

[6] A *self-eviction* is an eviction performed by the task itself. This can happen whenever the task footprint is larger than the cache size.

[7] The expression may in general be exponential in the parameters of τ; however, it is pseudo-polynomial if the system utilization is a priori bounded from above by a constant less than one.