Technical Report

Job Phasing Aware Preemption Deferral

José Marinho

Stefan M. Petters

HURRAY-TR-111203

Version:

Date: 12-05-2011

Job Phasing Aware Preemption Deferral

José Marinho, Stefan M. Petters

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail:

http://www.hurray.isep.ipp.pt

Abstract

Preemptions account for a non-negligible overhead during system execution. There has been substantial amount of research on estimating the delay incurred due to the loss of working sets in the processor state (caches, registers, TLBs) and some on avoiding preemptions, or limiting the preemption cost. We present an algorithm to reduce preemptions by further delaying the start of execution of high priority tasks in fixed priority scheduling. Our approaches take advantage of the \emphfloating non-preemptive regions model and explore the fact that, during the schedule, the relative task phasing will differ from the worst-casescenario in terms of admissible preemption deferral. Furtehrmore, approximations to reduce the complexity of the proposed approach are presented. Substantial set of experiments demonstrate that the approach and approximations improve over existing work, in particular for the case of high utilisation systems.

Job Phasing Aware Preemption Deferral

José Marinho and Stefan M. Petters
CISTER-ISEP
Polytechnic Institute of Porto
4200-072 Porto, Portugal
{*jmsm,smp*}*@isep.ipp.pt*

***Abstract*—Preemptions account for a non-negligible overhead during system execution. There has been substantial amount of research on estimating the delay incurred due to the loss of working sets in the processor state (caches, registers, TLBs) and some on avoiding preemptions, or limiting the preemption cost. We present an algorithm to reduce preemptions by further delaying the start of execution of high priority tasks in fixed priority scheduling. Our approaches take advantage of the *floating non-preemptive regions* model and explore the fact that, during the schedule, the relative task phasing will differ from the worst-case scenario in terms of admissible preemption deferral. Furtehrmore, approximations to reduce the complexity of the proposed approach are presented. Substantial set of experiments demonstrate that the approach and approximations improve over existing work, in particular for the case of high utilisation systems.**

I. Introduction

In today's technology, a vast majority of the processors deployed are not built into desktop or server computing systems, but instead are embedded into devices where the electronics enabled computations are not the core functionality. Besides the reliability and safety requirements, a class of those embedded systems, termed *real-time* systems, are subject to additional timing constraints. In this class correctness of an operation depends not only on its logical correctness, but also on the time of completion.

Preemptive schedulers, compared to non-preemptive ones [6], [9], [17], introduce time-overheads during the execution of the system due to context switches and the loss of working sets in the caches etc. generated by some preempting task [16], [1], [14].. These overheads are generally taken as null or negligible in scheduling theory, but are in fact substantial. This is tied to the inherent complexity of estimating the Cache-Related Preemption Delay (CRPD) and also tightly bounding the number of preemptions in fully preemptive systems.

A way to ease the task of quantifying preemption overhead is to introduce restrictions on the preemptions. This might be in the form of setting fixed preemption points to enable a tighter bound on CRPD [1] or dynamically delaying the preemption in the hope of completing the executions accordingly [19].

Two methods may be exploited in this manner. The first, *fixed non-preemptive regions*, relies on specific preemption points inserted into the task's code. This has to be done at off-line, relying on WCET estimation tools that can partition the task into non-preemptible sub-jobs [2]. This is highly restrictive since these points can not be replaced on run-time and it is non-trivial for complex control flow graphs. The second one *floating non-preemptive regions*, could be implemented by disabling preemptions for a certain amount of time or by restricting the dispatcher so it would not perform a context switch for a given time interval, still ensuring schedulability. This approach solely relies on the computation of the maximum admissible preemption deferral times. These can be changed on run-time if the task-set changes. The presented work will only address the *floating non-preemptive regions* model. This is the only model suited for use in multi-mode scenarios [15], where task-sets may vary at run-time. It also has the distinct advantage that no code changes are required in the application in order to implement it.

In this paper we investigate the improvement on the number of preemptions by dynamically delaying preemptions, exploiting the maximum admissible blocking times (or as we term it maximum admissible preemption deferral) and by taking advantage of the existing relative task release phasing on run-time. The effect of this is twofold. Firstly, the task to be preempted may complete the execution during this delay. Secondly, the arrival of further tasks during the delaying preemption phase also reduces the number, by leading to an ordered batch processing of all of these requests in priority order.

In the following section we introduce the system model. Section III is dedicated to the related work. Our methodology is described in Section IV before evaluating the approach in Section V. The final Section is devoted to concluding the work and indicating the directions of future work.

II. System Model

In this paper a task-set defined as a set $\tau = \{\tau_1, \ldots, \tau_n\}$ composed of n tasks is considered. We assume fixed task priority assignment where the element's index encodes the priority and fixed priority scheduling with floating non-preemptive regions. The task τ_1 holds the highest priority and τ_n the lowest. The set represented by $hp(i)$ denotes the set of tasks of higher priority than τ_i, which may be defined as $hp(i) = \{\tau_1, \ldots, \tau_{i-1}\}$. Each task is characterized by the three-tuple $\langle C_i, D_i, T_i \rangle$. The parameter C_i represents the worst-case execution time of each job from τ_i, D_i is the relative deadline and T_i the (minimum) distance between consecutive job releases in the periodic or sporadic model respectively. In fact our solution assumes sporadicity in the arrival pattern (i.e. each task τ_i may release a potentially infinite sequence of jobs separated by at least T_i time units) of jobs and constrained deadlines (i.e. $D_i \leq T_i$).

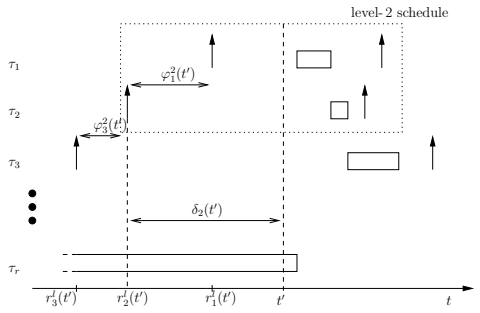

Figure 1. System Model Notation Clarification

Throughout the paper we consider $r_i^l(t)$ to represent the absolute time instant of the last release of a job of task τ_i with pending workload before (or at) time t. If there is no pending workload from task τ_i at time t then $r_i^l(t) = t$. The value δ_i^t denotes the amount of time elapsed since $r_i^l(t)$, i.e., $\delta_i^t \stackrel{\text{def}}{=} t - r_i^l(t)$. Then, let us define by $\varphi_j^i(t) = r_j^l(t) - r_i^l(t)$ the *task-relative* offset of task τ_j (in relation to task τ_i at time t), and by $\boldsymbol{\varphi}^i(t)$ the set of all $\varphi_j^i(t)$ such that $\tau_j \in hp(i)$. We address $\boldsymbol{\varphi}^i(t)$ as the vector of the offsets of higher priority releases of tasks in relation to τ_i. To clarify, this means that all the offsets are considered in relation to $r_i^l(t)$ which is the time instant of the last release of a job from task τ_i. If $r_j^l(t) < r_i^l(t)$ then the last release of τ_j, that still has pending workload at time t, preceded the last release of τ_i with pending workload at time instant t. If there is no pending workload from task τ_j at time t, this implies $\varphi_j^i(t) = \delta_i^t$. Whenever this is the case then no job from task τ_j is deferring its preemption at time instant $r_i^l(t) + \delta_i^t$ and that we have no knowledge about future releases. The stated notations are clarified in Figure 1.

III. Related Work

The mechanism of preemption deferral has a number of advantages as has been pointed out in several works [19], [18], [2]. These scheduling policies present a trade-off between the extremes of fixed priority non-preemptive and fully preemptive scheduling. Gang Yao et al. provide a comparison of all the available methods described so far in literature [19]. Non-preemptive scheduling has its benefits. Besides completely removing the problem of preemption delay, it schedules some task-sets that wouldn't otherwise be schedulable under fully preemptive fixed priority (FP) [17] and enables considerable memory savings by allowing for the existence of a single stack of size equal to the maximum stack requirement by any task that compose the task-set.

Slack computation was the subject of some attention in the past [7], [11]. These works mainly deal with the detection of slack in the schedule that enables the execution of aperiodic tasks with low priorities to execute uninterrupted. This benefits the latency of those applications considerably. The drawback of these methods is that they either rely on offline analysis and the periodic behaviour of hard deadline tasks [11] or on the on-line computation of the slack using methods with variable execution (recursive method) time and high complexity [7]. The previously mentioned works do not address the issue of preemption reduction nor consider slack stealing on a purely hard real-time system.

Wang and Saksena provided a hybrid of fully preemptive fixed priority scheduling with non-preemptive scheduling [17]. In this work a task may only preempt another if its priority is bigger than that task's preemption threshold. As a result some tasks will be unable to preempt other tasks despite having the higher priority. The preemption thresholds are computed by aid of a search algorithm that will test several possibilities until it either reaches a solution that ensures schedulability for the given task-set or fails.

A similar point of view is proposed by Fohler et al. [8]. In this work an off-line analysis parses the schedule identifying preemptions. It then tries to remove these by changing by changing tasks priorities and the offsets without jeopardizing the schedulability. This method is only aplicable to periodic tasksets though.

Keskin et al. discuss the theory of deferred preemption schedulability [10]. The author deemed the available test [6] optimistic, arguing that under no assumptions the worst-case response time for a job of task τ_i may no longer arise in the first critical region in a synchronous release situation but that it may show up in a job k of task τ_i in the level-i active period generated at a synchronous release situation. This gives the indication that finding the worst-case situation for the deferred preemption fixed priority scheduling is not straightforward.

Gang Yao et al. [18] also provides a way to compute a bound on the size of the floating non-preemptive regions. This is computed using the request bound function. It basically derives the amount of idle time (β_i) for the critical region of task τ_i in a synchronous release situation. Task τ_i may then endure a delay of β_i. The length of the floating non-preemptive regions (represented by Q_i) are computed by: $Q_i = \min_{k \in hp(i)}(\beta_k)$. The way β_i is computed restricts its usage to situations where the task-set is schedulable under fully preemptive fixed priorities, though it effectively decreases the number of preemptions in relation to that scheduling policy, but does not take into account the task phasing.

Gang Yao et al. devised a fixed priority scheduling method [20] where a maximum bound on the length fixed non-preemptive regions is provided. In this situation the computed β_i's are generally larger than in the previous work [18] because the last chunck of a task's execution is not taken into account during the analysis. This enables a further reduction on the number of preemptions. Still this only allows the scheduling of tasks that were schedulable under the fully preemptive model and can only be applied for the fixed non-preemptive model. The author proves that if the task-set is schedulable under fully preemptive fixed priority, the job of task τ_i with worst-case response time will still be the first job in the synchronous release situation. This schedulability test has the intention of having a lower complexity than Keskin's solution [10].

Reducing the number of preemptions helps on decreasing the level pessimism added to the schedulability test. Staschulat and Ernst provided a method to estimate the CRPD for instruction cache in [16], Ramaprasad and Muller have solved the problem of estimating CRPD for data caches to some

extent [13]. This area has been constantly extended in works of Altmeyer et al. [1].

Restricting preemption points presents a viable way to address the problem of preemption delay. In fully preemptive scheduling whenever a release occurs, preemption immediately takes place if the job released is of higher priority than the running one. Ramaprasad derives a method based on the knowledge of the response time of higher priority tasks and their periods to bound the number of feasible preemption points [14]. This enables the designer to have a less pessimistic view on preemption delay since normally contiguous line codes have the same preemption cost.

A more accurate way of doing so exploits the insertion of fixed preemption points into tasks, using the models described by Burns [6] (*fixed preemption points*). This procedure exploited by Bertogna et al. [2] has the limitation of only being suited for fixed preemptive regions. Exploiting the reduction of preemptions by using online information to increase the floating non-preemptive region length was not addressed until this point.

IV. Methodology

A. Admissible Preemption Deferral

Every task can endure a preemption deferral which solely depends on the amount of higher priority workload that will need to be executed in the future, as will be shown later in this section. We first introduce a few concepts taken from related work.

Definition 1 (level-i schedule)**.** *The schedule composed of jobs from task τ_i and jobs from tasks with higher priority is denominated as* level-i *schedule [10].*

The reader may find an example for term level-i in use in the graphical representation provided in Figure 1. We compute the amount of idle time that will exist in the level-i schedule in a specific time interval , as a function of the known previous higher priority releases that are still deferring.

$$W_i(t, \boldsymbol{\varphi}^i(t)) \stackrel{\text{def}}{=} C_i + \sum_{j \in hp(i)} \text{rbf}(\max(t - \varphi_j^i(t), 0), \tau_j) \quad (1)$$

where

$$\text{rbf}(t, \tau_j) \stackrel{\text{def}}{=} \left\lceil \frac{t}{T_j} \right\rceil \times C_j. \quad (2)$$

Equation 1 [12] gives us the amount of pending workload in the level-i schedule that was released up until time instant t and is being deferred (if higher priority constraints enable so). By computing the difference between Equation 1 and the time progression line for every point in the given time interval $[0, \delta_i^t]$ and chosing the maximum of such values, we have a quantification of the amount of idle time in the schedule for the specified time interval. This may be formaly written as

$$B_i(\delta_i^t, \boldsymbol{\varphi}^i(t)) \stackrel{\text{def}}{=} \max_{t' \in [0, \delta_i^t]} (D_i - t' - W_i(D_i - t', \boldsymbol{\varphi}^i(t))). \quad (3)$$

The intuition behind the function 3 is depicted in Figure 2 for a level-3 schedule. Figure 2 is composed by three plots, the

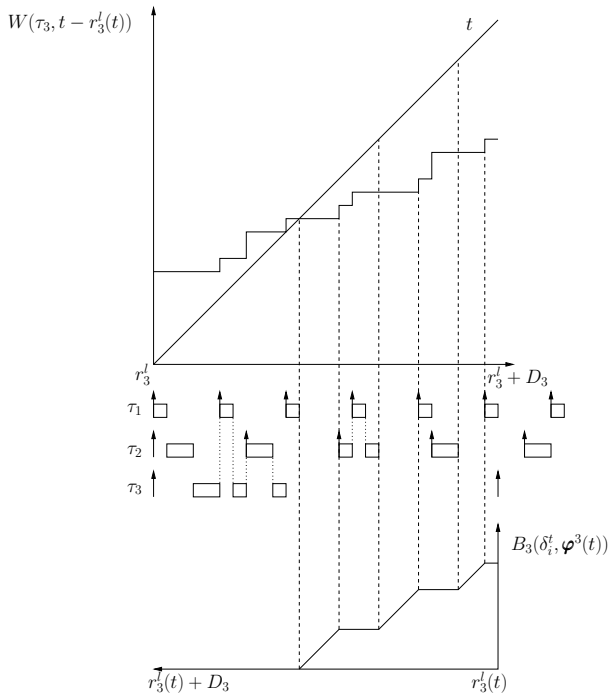

Figure 2. Computation of Equation 3 with no Known Prior Higher Priority Releases

beginning of the referential is $r_3^l(t)$ which is the time instant of the release of the job of τ_i considered in this example. Without loss of generality $r_3^l(t)$ may equate to 0 (i.e. $r_3^l(t) = 0$). The top graph depicts Equation 1 on the same plot with the time progression line. The maximum difference between the time line and the function defined by Equation 1 at every interval $[0, \delta_i^t]$ (where $t \in [r_i^l, r_i^l + D_i]$) gives the amount of time there was no pending workload in the system to be processed in that same interval. This is apparent by observing the schedule chart in the middle of Figure 2. The bottom part plot in Figure 2 displays the amount of idle time available in the level-3 schedule at the release time of a job from τ_3 and its evolution from when $\delta_i^t = 0$ as it progresses towards $\delta_i^t = D_i$, as task's τ_3 workload gets deferred and the higher priority workload shifted for δ_i^t time units as well. From the point of view of the current active job of task τ_3, as it is deferring its preemption in time, the amount of time that it can endure to be further deferred at every instant δ_i^t is obtained by computing Equation 3 along the time line of deferral. In the same figure a situation where no higher priority releases have occurred prior to $r_3^l(t)$ is displayed intentionally in order to provide a clearer example of the computation of Equation 3 and its evolution with time. In this specific scenario it is easily perceivable that if all higher priority workload is shifted to the right in conjunction with τ_i's job the available idle time will itself "shift" in the same manner, hence the evolution with time of the bottom plot.

Theorem 1. *After the release of a job of τ_i at time $r_i^l(t)$, if some lower priority task is executing the preemption may be safely deferred for $B_i(\delta_i^t, \boldsymbol{\varphi}^i(t))$ time units, without jeopardizing τ_i's deadline.*

Proof: At time instant $r_i^l(t)$ there will be at least $B_i(D_i, \boldsymbol{\varphi}^i(t))$ time units of idle time in the level-i schedule up until $r_i^l(t) + D_i$. If the level-i preemption is deferred for ϵ time units then $B_i(\delta_i^t, \boldsymbol{\varphi}^i(t)) - \epsilon$ time units of idle time would be available for the level-i schedule at time $> r_i^l + \epsilon$. At the earliest time instant t'' which makes $B_i(\delta_i^{t''}, \boldsymbol{\varphi}^i(t)) = 0$ no more idle time will be available in the level-i schedule until $r_i^l(t) + D_i$. From $r_i^l(t)$ to t'' the task may be safely delayed since there will always be enough time to execute completely before its deadline even if fully preemptive schedule would be carried out from this point onwards. ■

The previous Theorem handles a generic case of what was shown in [4] for the synchronous arrival of higher priority workload situation. Note that Theorem 1 only refers to the amount of time a job of task τ_i may be deferred so that it doesn't miss its deadline. The same reasoning has to be applied to all jobs currently deferring the preemption so that no deadline is missed in the system. At every instant in time, while there are jobs deferring their preemptions $B_i(\delta_i^t, \boldsymbol{\varphi}^i(t)), \forall i \in S$ is computed, where $S = lep(j) \cap hp(p)$ In this case τ_j denotes the highest priority defering at time t, τ_p the task currently running and $lep(j)$ is the set of task of lower or equal priority in relation to task τ_j. At the time instant when there exist an i such that $B_i(\delta_i^t, \boldsymbol{\varphi}^i(t)) = 0$, a preemption occurs and all previously deferring jobs cease to defer, at which point the job with highest priority deferring is scheduled onto the processor. Normal fixed priority scheduling is carried out until there is a release of a task of higher priority than the currently running task. Implementing a scheduling policy following this exact methodology is clearly unrealistic since it implies a high complexity algorithm to operate at every instant in time. Some approximations may be used though. The simplifications rely on the observations described in the following text.

B. Decreasing complexity

A straightforward way to exploit the knowledge provided by Equation 3, is to trigger a non-preemptive execution region for the job of task τ_l currently running, whenever a release from a higher priority job happens. In the next step we consider the situations where no higher priority job releases are present in the system so all the elements in vector $\boldsymbol{\varphi^i}(t)$ will be equal to the amount of time elapsed since $r_i^l(t)$. Equation 3 may then be rewritten as,

$$B_i(\delta_i^t, \boldsymbol{\varphi}^i(t)) = B_i(\delta_i^t), \tag{4}$$

since $\forall j \in hp(i), \varphi_j^i(t) = \delta_i^t$. The non-preemptive region should have a duration equal to $\min_{i \in hp(l)}(B_i(0))$. This approach is the one presented by Gang Yao et al. [18]. A simple low complexity build up on the previous approach would be to still consider $B_i(0)$ as the time a job from task τ_i may defer its preemption, and create a set of rules that enable the scheduler to perform better or in the same manner whenever the schedule is not in the worst-case scenario (synchronous release of higher priority tasks).

Property 1. *Whenever a job of task τ_i is released, if it has higher priority than the running task and no other job is already deferring its preemption, the scheduler may safely delay its preemption by $B_i(0)$).*

This stems from the fact that if no higher priority was released before the amount of idle time available on the level-i schedule will always be greater or equal to $B_i(0))$ in the critical region of this job. So it may safely be delayed for that amount and always complete before the deadline as stated in Theorem 1.

Property 2. *If one or more tasks are already deferring their preemption and no job has higher priority than the job from τ_i, the timer is set to* $\min(timer, B_i(0))$.

The completion before deadline of one job only depends on the higher priority workload. The previously deferring jobs already set the timer in order not to miss the deadlines. If the lower priority tasks are to complete execution before deadline the minimum amount of time that all jobs may be deferred for has to be taken into consideration in this situation.

Property 3. *If in the previous situation there is at least one higher priority job already deferring its preemption, the timer is set to* $\min(timer, \max(B_i(0) - (r_i^l(t) - t_0), 0)$*, where t_0 is the instant in time when the first job that started the current preemption deferral thread arrived.*

This is due to the fact that $B_i(0)$ represents the amount of time a job may be deferred if no other higher priority job is deferring at that instant in time. If at time $r_i^l(t)$ there is higher priority workload deferring preemption it was released mandatorily after t_0 (i. e. $t_0 < r_i^l(t)$. This implies that at time instant t_0 no job with higher priority than the current job of task τ_i was deferring its preemption. If the current job of task τ_i had been released at time t_0 it could defer its preemption until $t_0 + B_i(0)$ without missing its deadline, as a consequence if the job arrives at a time instant after t_0 it can still defer its preemption until the same point in time $(t_0 + B_i(0))$.

C. Overload Situation

When trying to leverage these three properties we must make sure that the following overload situation does not occur. A task may be released in a situation where the admissible preemption deferral would in fact be negative. In this situation a deadline may be missed. During the schedulability test a syschronous release situation is considered. This was proven to be the situation leading to the worst-case respose time of a task in fixed priority scheduling [5]. In restricted preemption fixed priority scheduling policy the synchronous release of higher priority tasks situation may not lead to the largest response time of a task [5].

Lemma 1. *No more than one additional job from every higher priority task may appear in the time interval bounded by a release and a deadline of a task.*

Proof: In the synchronous release situation $\left(\left\lfloor \frac{D_i}{T_j} \right\rfloor + 1\right) \times C_j$ *units of higher priority workload per higher priority task may be considered. If some lower priority task deferes the start of execution of an higher priority task then no more than* $\left\lceil \frac{T_i + D_i}{T_j} \right\rceil \times C_j$ *units of workload need*

to be considered. If a middle priority task τ_i had a release more than T_j time units after the release of a job from task τ_j then, considering the constrained deadline task model, the workload of task τ_j would have been concluded at the time of release of task τ_i, hence not interfeering in its response time. ■

As a consequence of Lemma 1 a sufficient schedulability test ensuring that no such situations can occur is presented in Equation 5.

$$D_i \geqslant C_i + \sum_{j \in hp(i)} \left\lfloor \frac{D_i}{T_j} \right\rfloor \times C_j + \sum_{j \in hp(i)} \left(\min(2 \times C_j, D_i - \left\lfloor \frac{D_i}{T_j} \right\rfloor \times T_j \right) \quad (5)$$

This condition reflects the fact that at most one aditional job of every higher priority task may be present in the time interval bounded by a release and deadline of a middle priority job. Equation 5 takes into account the workload of every higher priority task that executes entirely untill completion in a time interval of length D_i and then sums two aditional workloads or the length of the interval $D_i - \left\lfloor \frac{D_i}{T_j} \right\rfloor \times T_j$, whichever is minimum, as aditional higher priority workload. For tasksets that miss the sufficient schedulability test a safety mechanism needs to be added to the protocol, that prevents deadlines from being missed. At run-time the sum of the worst-case execution time of the task with higher priority than the running task are quantified. If at a release this value becomes greater than $\min_{j \in hp(i)}(B_j)$ then the timer is set to $t - t_0 + \min_{j \in hp(i)}(B_j)$. The previously described set of rules takes into account the existence of job release phasing in relation to the worst case (synchronous release of higher priority) during the normal run of the schedule and enables better decisions when sporadic behaviour is present (commonly denominated as minimum inter-arrival time). The set of rules may be enhanced by considering that if no higher priority workload is available then the job may be even further delayed. This fact motivates the following theorem.

Theorem 2. *A job of task τ_i may defer its preemption for $D_i - \mathrm{WCRT}_i$ time units while no job with priority higher than τ_i is released in the time interval $[r_i^l(t), r_i^l(t) + \Delta_i]$.*

Proof: We term the amount of preemption deferral time for Δ_i defined as $\Delta_i \stackrel{\text{def}}{=} D_i - \mathrm{WCRT}_i$. The quantity WCRT_i is defined in the usual way, $\mathrm{WCRT}_i = R^k$, where k is the smallest value that satisfies $R^k = R^{k-1}$. The R_k value is iteratively computed by the following equation [6],

$$R^k = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R^{k-1}}{T_j} \right\rceil \times C_j \quad (6)$$

and choosing $R^0 = C_i + \sum_{j \in hp(i)} C_j$ as the initial value in the iteration for R. If no higher priority job is released in the interval $[r_i^l(t), r_i^l(t) + \Delta_i]$, the job from task τ_i meets its deadline if it preempts at $t = r_i^l(t) + \Delta_i$ irrespective of the preemption deferral admissible for all the higher priority workload that may be released after or at instant $r_i^l(t) + \Delta_i$, due to the definition of WCRT_i. ■

Using the previous theorem in the protocol generates a situation requiring special consideration: Assume one task is deferring, having set the timer on arrival to its corresponding Δ according to the previous theorem. A higher priority task arriving may not use the past value in the timer, but needs now to take into account the $B(0)$ of the highest priority job already deferring.

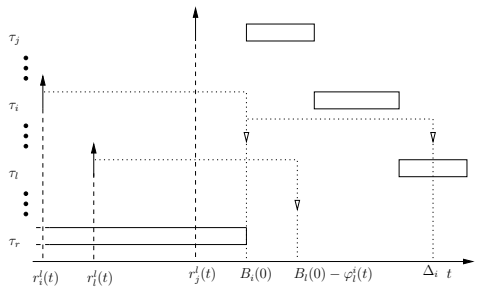

Figure 3. Scenario Motivated by Theorem 2

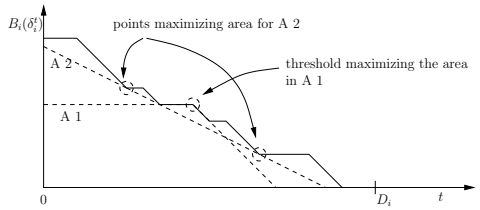

Figure 4. Outline of the Devised Approximations for Equation 3

An example of a problematic situation is displayed in Figure 3. If some middle priority job of task τ_i is released at time $r_i^l(t) = 0$, followed by a release of a lower priority job from task τ_l, $\varphi_l^i(r_l^l(t))$ time units after, the situation depicted in Figure 3 may arise. The hollow arrows in the picture represent timer values that tasks contended with in the timer setting procedure and set the timer (i.e. were the minimum value at the contending time).

According to previous rules the job from task τ_l sets the timer since its admissible preemption deferral is smaller than the value τ_i loaded into the timer, $\max(B_l(0) - r_l^l(t)), 0) < \Delta_i - r_l^l(t)$. At time instant $r_j^l(t)$, when a job from task τ_j was released (where j is of higher priority than i) τ_i has again to check if its admissible deferral time is the minimum among all deferring jobs. When it was released it tried to set the timer until Δ_i, since higher priority workload is available and no knowledge about Equation 3 exists at this stage apart from $B_i(0)$, the scheduler has to check if this would yield the minimum value for the timer if it was used at time $r_i^l(t)$. When this scenario occurs the following relation is true $\max(B_l(0) - r_j^l(t), 0) > \max(B_i(0) - r_j^l(t), 0)$. The timer has to be loaded at this time with $\max(\min(B_l(0) - \varphi_j^i, B_i(0) - \varphi_j^i, \Delta_j), 0)$. The information on the highest priority job deferring its preemption has to be stored then, along with the time instant of its release in order for the mechanism to work, since when some other higher priority workload is released its admissible deferral has to mandatorily be reevaluated.

D. Admissible Deferral Approximation

It is clear that a trade-off between memory usage and efficiency may be exploited in order to use better the knowledge presented by Equation 3. Using only its initial point is too restrictive. We present two efficient approaches for having a

lower bound on $B_i(a)$ that may be used at run-time to achieve longer preemption deferrals.

A 1: one may consider the usage of another point of $B_i(a)$ function to aid on better deferral decisions. Based on the notion that $B_i(a)$ is monotonically decreasing, and that $\frac{dB_i(a)}{dt} = 0$ or $\frac{dB_i(a)}{dt} = -1$ the following method may be devised. The value to be used is $B_i(\text{threshold})$, meaning that any job of task τ_i may be deferred for $B_i(threshold)$ if no $threshold$ time units have elapsed since its release. After $threshold$ time units have gone by since the release the value is set to $B_i(threshold) - (threshold - timer)$. This approximation is made possible by the specificities of Equation 3 previously referred.

A 2: a linear equation which is a lower bound of $B_i(a)$ in the interval $[0, \Delta_i]$ may be created. This linear function is always smaller or equal to $B_i(a)$ and tangent to the convex hull defined by $B_i(a)$. At the time of release of a higher priority task relative to the previously highest deferring task the deferral is computed by $\frac{y_2-y_1}{x_2-x_1} \times a + (y_1 - \frac{y_2-y_1}{x_2-x_1} \times x_1)$. Both quantities $\frac{y_2-y_1}{x_2-x_1}$ and $y_1 - \frac{y_2-y_1}{x_2-x_1} \times x_1$ are computed offline.

Both methods define a lower bound on $B_i(a)$ function which takes up little memory space and may be exploited quite efficiently. Without any prior knowledge on the arrival pattern of higher priority workload and possible phasings a rule of thumb stating that both areas should be maximized to achieve better performance should be used. For the first approach (A 1), the point that maximizes $\text{area} = a \times B_i(a) + \frac{a^2}{2}$ is the one chosen. The second one (A 2) chooses the two adjacent points defined by (x_1, y_1) and (x_2, y_2) of the convex hull defined by function $B_i(a)$ that maximize $\text{area} = \frac{y_1^2}{m} + 2 \times y_1 \times x_1 - m \times x_1^2$, where $m = \frac{y_2-y_1}{x_2-x_1}$. Both approximation are used in a scenario where the previous highest priority job deferring its preemption faces a release from a higher priority job.

E. Implementation Overhead

All of the methods presented so far have small complexity, having at maximum three comparisons when setting the timer. At every release a maximum of four values have to be compared in order to chose the minimum. The last approximations rely on a limited number of computations, as was shown in A 1 and A 2 description. These computations are cheap in comparison to the overall savings allowed for them.

F. Tighter Bound on Preemption Number

The maximum number of preemptions per task may be upper bounded both in the state of the art [18] as well as in our methods by $\left\lfloor \frac{C_i}{Q_i} \right\rfloor$, where $Q_i = \min_{j \in hp(i)}(B_j(0))$. This bound is implicitly stated in the work of Gang Yao et al. [18]. For our method it suffices to state that whenever a job from task τ_i executes on the processor it will do so for at least Q_i time units uninterrupted by a higher priority workload. Observe that in all the presented protocols whenever an higher priority task τ_j release occurs a the lower priority task will still execute non-preemptively for at least $B_j(0)$. If subsequent higher priority releases occur, for which the corresponding tasks have smaller $B_k(0)$ (i.e $B_k(0) < B_j(0)$) in the worst case scenario then $B_k(0)$ time units would be counted since t_0 (the start of the deferral chain). If $B_k(0) = \min_{j \in hp(i)}(B_j(0))$, then $Q_i = B_k(0)$. Hence the same bound applies. Suppose the following taskset The value denoted by ϵ is a small as

	C_i	T_i	Q_i
τ_1	2	5	∞
τ_2	3-ϵ	7	3
τ_3	2	5	ϵ

Table I
EXAMPLE TASK-SET

needed quantity. This yields the following relation,

$$\lim_{\epsilon \to 0} \left(\left\lfloor \frac{C_i}{\epsilon} \right\rfloor \right) = \infty. \tag{7}$$

The bound provided, considering the previous bound for jobs of task τ_3 would be overly pessimistic in cases where Q_i is small. Bear in mind that this is an extreme case to motivate the fact that, in specific situations, the number of higher priority releases in a given interval is itself a tighter bound on the number of preemptions for a given task. All the higher priority jobs that arrive $\text{WCRT}_i - Q_i$ time units after the job of τ_i started first to execute are guaranteed not to preempt and hence may be disregarded in the maximum number of preemption computation. This indicates that a suitable bound should then be

$$\min \left(\left\lfloor \frac{C_i}{Q_i} \right\rfloor, \max \left(\sum_{j \in hp(i)} \left\lceil \frac{\text{WCRT}_i - Q_i}{T_j} \right\rceil, 0 \right) \right). \tag{8}$$

V. EVALUATION

In this section comparative results on all the approaches presented in this paper are showcased. The *first approach* is the one that implements Properties 1 to 3 . The *second approach* implements the method described on the Theorem 2 on top of the *first approach*. The *third* and *fourth approaches* implement the approximation of Equation 3 on top of the *second* one.

A. Discussion

By further delaying higher priority workload a growing number of releases will be merged and the subsequent thread of jobs will execute in the correct priority order from the beginning without the need for preemptions. Parallel to that mechanism, delaying further also enables higher priority workload to wait for a lower priority job to finish its execution, hence reducing the number of preemptions as well. Both these facts contend with a contrary effect. By further delaying some middle priority jobs, situations where hypothetical higher priority jobs arrive and cannot be deferred for the same amount of time will generate preemptions where none should have existed if all jobs were deferred for the same amount of time as a function of the priority of the running task (Gang Yao et al. approach [18]). Our claim is that the first two effects generally dominate over the third one. This is supported by the experimental data presented in the following subsection. The possibilities of increasing the number of preemptions in relation to Gang Yao only stems from the fact that higher priority workload is being moved in the schedule, this does not change any of the off-line guarantees in terms of preemptions for each task.

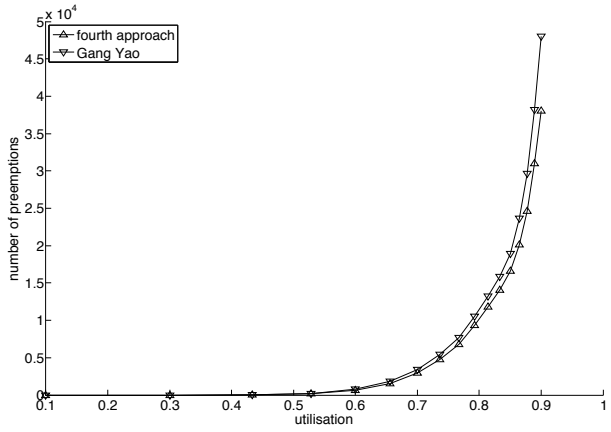

Figure 5. Implicit Deadline Model - 16 tasks

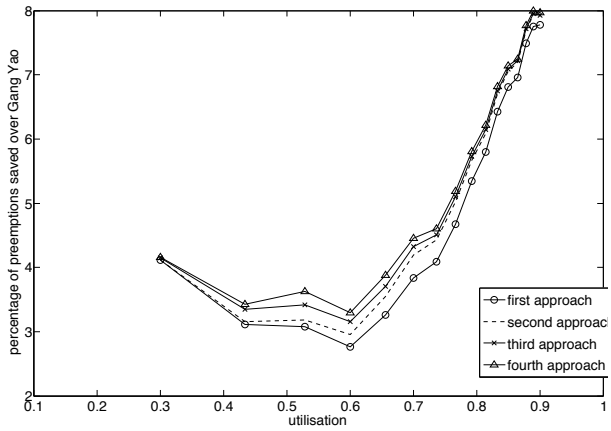

Figure 6. Implicit Deadline Model - 4 tasks

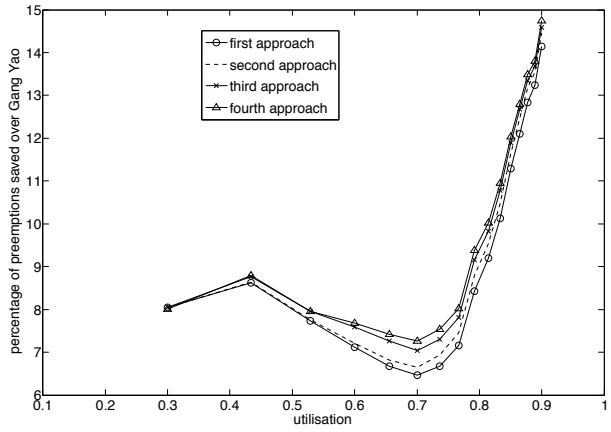

Figure 7. Implicit Deadline Model - 8 tasks

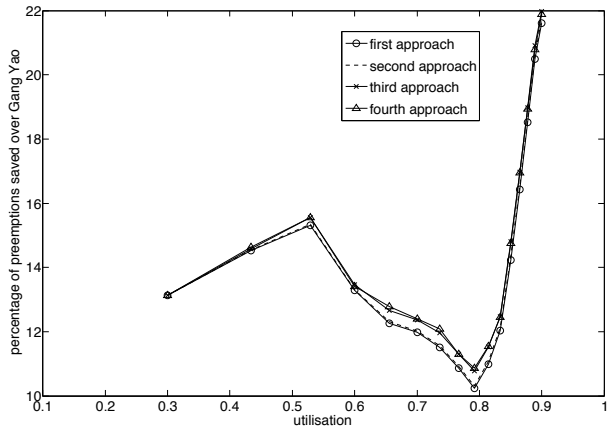

Figure 8. Implicit Deadline Model - 16 tasks

B. Simulations

The three system models were evaluated using simulations. In each model all tasks are generated using the unbiased task-set generator method presented by Bini (UUniFast) [3]. Tasks are randomly generated for every utilization step, their maximum execution requirements (C_i) were uniformly distributed in the interval $[50, 500]$. In the first situation the task-set behaves in a fully periodic manner with implicit deadlines ($D_i = T_i$).

In the second situation constrained deadlines are investigated. The constrained deadline model was implemented by randomizing the period of the tasks in relation to their deadlines. For this data run the periods are constructed in the following manner $D_i = T_i - S$, where S is a random variable with uniform distribution in the interval $[0, 0.2 \times T_i]$. In the sporadic model the consecutive release of a task is $T_i + A$ units separated from the last release of the same task. A is taken from a uniform distribution in the interval $[0, 0.5 \times T_i]$.

On every utilization step the schedule of 10000 fixed priority feasible tasksets is simulated and subsequently the preemption number is averaged across all runs. Utilization steps are non-uniformly distributed and the points are given by the function $\text{step}(k) = 1.1 - \frac{1}{1+\frac{1}{1+k\times 4}}$ where $k \in [0, 16]$. This enables us to get a better concentrations of data at higher utilisations.

1) Implicit Deadlines Periodic Model: In Figure 5 the results in number of average preemptions for task-sets with 16 tasks are presented showing the state of the art algorith and our fourth approach. Both lines display an exponential behaviour, the fourth approach has a brief offset, which imply big preemptions savings as we show in the following plots. It is generally observable that the approaches proposed in this work outperform the state of the art in number of avoided preemptions in the schedules in particular at higher utilisations as is show in Figures 6 to 8.

As the number of tasks increase we can also observe in Figures 6 to 8 that the gains of our approach in comparison to Gang Yao's method become even more evident. This is tied to the fact that with more tasks there will be more situations where the gains, in terms of admissible deferral time, allowed by task phasing appear. It is worth noting that there is a considerable number of preemptions that can not be avoided, the higher priority jobs that are released while some lower priority workload is executing and that have their deadline inside the response time of the lower priority workload will always have to preempt it. The displayed data does not make a distinction between unavoidable preemptions and the ones that might possibly be avoided by delaying preemptions a bit more in a feasible way. For the implicit deadline case with no sporadicity at the biggest utilisation point tested when the task-set is composed of 16 tasks, roughly 21% of the total number of preemptions yielded by the state of the art method are saved.

2) Constrained Deadlines: By introducing constrained deadlines the off-line assumptions about maximum deferral

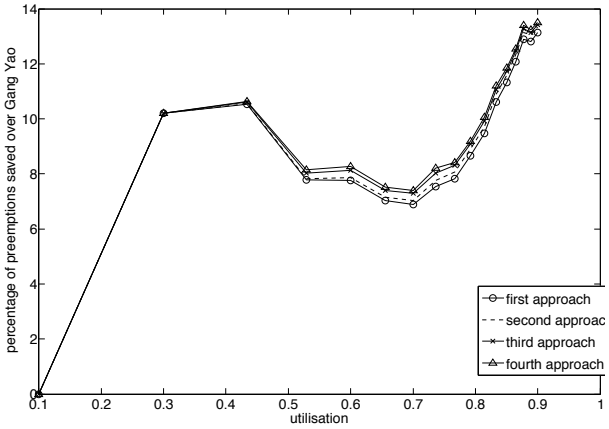

Figure 9. Constrained Deadlines Model - 8 tasks

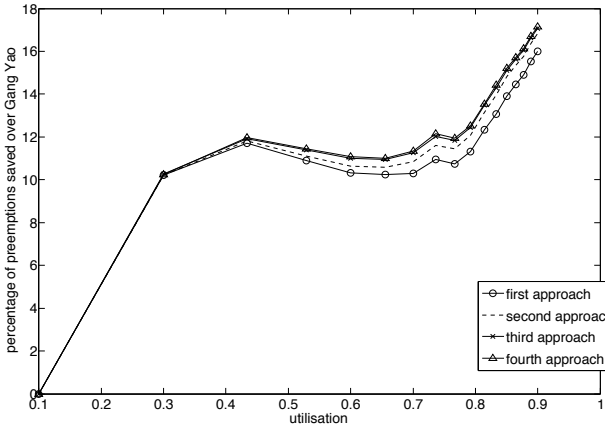

Figure 10. Sporadic Model - 8 tasks

time are drastically reduced and hence the opportunities for online phasing exploitation are increased, in particular for lower utilisations. In Figure 9 the gains of the four approaches are compared against the state of the art method. While only the results for the task set with 8 tasks are shown, the other task-set sizes expose similar tendencies.

3) Sporadic Behaviour: Similar to the constrained deadlines model, a shift towards gains at lower utilisations can be observed in Figure 10. This can be explained with the reduced actual workload due to sporadicity and the increased scope for exploiting online information. Somewhat counterintuitively these additional gains are not apparent at very high utilisations. Here we again only depict the results for 8 tasks, however, the other task-set sizes were also exposing similar trends.

VI. Conclusions

We have presented a series of approaches that reduce the number of preemptions in fixed priority using floating non-premptive regions. Each proposed approach incrementaly extends the length of the admissible non-preemptive region by exployting the task phasing in the schedule using on-line mechanisms whith extremely low complexity. We compared all the proposed solutions in this paper with the state of the art by Gang Yao [18]. By running an extensive set of simulations we show that our methods surpass the state of the art in average number of preemptions. The data presented also shows that the gains increase at high utilisations and with bigger task-sets. All of the solutions have a small implementation overhead, by enabling some considerable savings in preemption count they prove to be a relevant method for task-set scheduling. Furthermore we have also shown that for some situations the available bound is a crude estimation on the maximum number of preemptions a task may endure and present an enhancement. As future work we intend to extend these results for dynamic task priority and provide a tighter bound on the number of maximum preemptions.

References

[1] S. Altmeyer, C. Maiza, and J. Reineke. Resilience analysis: tightening the CRPD bound for set-associative caches. In *LCTES'10*, pages 153–162, 2010.

[2] M. Bertogna, G. Buttazzo, M. Marinoni, G. Yao, F. Esposito, and M. Caccamo. Preemption points placement for sporadic task sets. In *22nd ECRTS*, pages 251 –260, Jul 2010.

[3] E. Bini and G. Buttazzo. Biasing effects in schedulability measures. In *16th ECRTS*, pages 196 – 203, Jun 2004.

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

[5] R. Bril, J. Lukkien, and W. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited. In *Real-Time Systems, 2007. ECRTS '07. 19th Euromicro Conference on*, pages 269 –279, july 2007.

[6] A. Burns. Preemptive priority-based scheduling: an appropriate engineering approach. In S. H. Son, editor, *Advances in real-time systems*, pages 225–248. 1995.

[7] R. Davis, K. Tindell, and A. Burns. Scheduling slack time in fixed priority pre-emptive systems. In *Real-Time Systems Symposium, 1993., Proceedings.*, pages 222 –231, dec 1993.

[8] R. Dobrin, G. Fohler, and P. Puschner. Translating off-line schedules into task attributes for fixed priority scheduling. *Real-Time Systems Symposium, IEEE International*, 0:225, 2001.

[9] L. Georges, P. Muhlethaler, and N. Rivierre. A few results on non-preemptive real time scheduling. Research Report RR-3926, INRIA, 2000.

[10] U. Keskin, R. Bril, and J. Lukkien. Exact response-time analysis for fixed-priority preemption-threshold scheduling. In *ETFA2010*, Sep 2010.

[11] J. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Real-Time Systems Symposium, 1992*, pages 110 –123, dec 1992.

[12] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *10th RTSS*, pages 166–171, 1989.

[13] H. Ramaprasad and F. Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *12th RTAS*, pages 71–80, Apr 2006.

[14] H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemption points. In *27th RTSS*, pages 212–224, 2006.

[15] Y. Shin, D. Kim, D. Kim, and K. Choi. Schedulability-driven performance analysis of multiple mode embedded real-time systems. In *37th DATE*, 2000.

[16] J. Staschulat and R. Ernst. Multiple process execution in cache related preemption delay analysis. In *4th EMSOFT*, 2004.

[17] Y. Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *6th RTCSA*, pages 328 –335, 1999.

[18] G. Yao, G. Buttazzo, and M. Bertogna. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. In *15th RTCSA*, 2009.

[19] G. Yao, G. Buttazzo, and M. Bertogna. Comparative evaluation of limited preemptive methods. In *ETFA2010*, Sep 2010.

[20] G. Yao, G. Buttazzo, and M. Bertogna. Feasibility analysis under fixed priority scheduling withlimited preemptions. *Real-Time Systems*, pages 1–26, 2011.