Extending Fixed Task-Priority Schedulability by Interference Limitation

José Manuel Marinho*, Stefan M. Petters*, Marko Bertogna†
*CISTER-ISEP Research Centre, Polytechnic Institute of Porto, Portugal
†University of Modena, Modena, Italy
Email: {jmsm,smp}@isep.ipp.pt, marko.bertogna@unimore.it

***Abstract*—While the earliest deadline first algorithm is known to be optimal as a uniprocessor scheduling policy, the implementation comes at a cost in terms of complexity. Fixed task-priority algorithms on the other hand have lower complexity but higher likelihood of task sets being declared unschedulable, when compared to earliest deadline first (EDF). Various attempts have been undertaken to increase the chances of proving a task set schedulable with similar low complexity. In some cases, this was achieved, by modifying applications to limit preemptions, at the cost of flexibility. In this work we explore several variants of a concept to limit interference by locking down the ready queue at certain instances. The aim is to increase the prospects of schedulability of a given task system, without compromising on complexity or flexibility, when compared to the regular fixed task-priority algorithm. As a final contribution a new preemption threshold assignment algorithm is provided which is less complex and more straightforward than the previous method available in the literature.**

I. Introduction

In today's technology, a vast majority of the processors deployed are not built into desktop or server computing systems, but are instead 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 outcome, but also on the time of completion.

The scheduling policy used to carry out a given workload will greatly influence the temporal behaviour of the tasks in the system. The decision on which scheduling policy to use is an exercise where the trade-offs have to be carefully considered by the system designer. Scheduling disciplines may be weighted according to several metrics. An important one is schedulability (i.e. its ability to schedule task-sets). Increased schedulability guarantees generally come with the cost of increased complexity in the scheduling decisions. This may lead to unnecessary overheads due to scheduler operation, which must be upper bounded and taken into account in the system schedulability assessment. If a given task-set is schedulable with a lower complexity scheduling mechanism, then there might be little motivation for deciding to use a higher complexity one.

Another important aspect pertaining to system operation is the number of preemptions the tasks are subjected to. These overheads are generally taken as null or negligible in scheduling theory, but are in fact substantial.

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 Cache Related Preemption Delay (CRPD) [1] . Setting preemption points is an effective method to increase the schedulability of fixed task-priority systems [2]. The mechanism denominated *fixed non-preemptive regions*, relies on specific preemption points inserted into the task's code. This has to be done at design time, relying on worst case execution time estimation tools that can partition the task into non-preemptible sub-jobs [2]. This is highly restrictive since these points can not be replaced at run-time. Furthermore, the choice of preemption points placement proves to be a non-trivial task for complex control-flow graphs and flexibility, with respect to task-set changes, is as well often recommended.

Having a more flexible mechanism helps to reduce development, software maintenance, and update costs. It also facilitates the operation of systems which require run-time workload changes.

An example of a considerably more flexible limited preemptive mechanism is the *floating non-preemptive regions* model. In this model a non-preemptive region of execution is started once at the time instant t when a job of higher priority, than the currently executing job from task τ_i, is released where at time $t - \epsilon$ the job from task τ_i had the biggest priority from all the active jobs at that time instant. This non-preemptive region has a limited duration of Q_i time units.

This approach solely relies on the computation of the maximum admissible preemption deferral times for each priority level. This information can be swiftly changed at run-time if the task-set changes in order to adapt to the new workload timing properties. Even though the *floating non-preemptive regions* allows considerably more flexibility than *fixed non-preemptive regions* it can not be easily exploited in order to increase the schedulability of fixed task-priority scheduling policy for a sporadic task model.

In this paper we investigate the improvement on the schedulability of task-sets by limiting the interference suffered by lower priority tasks. This is done by introducing a new task parameter termed ready-queue locking time

instant. If a job from a task has pending workload at its ready-queue locking time instant then the ready queue is locked, preventing higher priority workload from being inserted into the ready queue and hence interfering with its execution. As a consequence, the upper bound on the number of preemptions each job might suffer is reduced in comparison to the regular fixed task-priority policy. A new preemption-threshold scheduling policy is provided so that ready-queue locking can be used together with the aforementioned mechanism. The proposed methods may straightforwardly be used in conjunction with the *floating non-preemptive regions* which further helps on the reduction of preemptions during workload execution.

The properties of the solutions provided in this work straightforwardly allow for on-line changes in the task-set, since they only require an update of the relative ready-queue locking time instant of every task. The complexity of the proposed solutions is much lower than optimal uniprocessor scheduling policies. Namely, running the ready-queue locking mechanism has a timing complexity of $O(1)$ associated to it, which is considerably better than the $O(n \times \log(n))$ associated to earliest deadline first.

In the following section we introduce the system model. Section III is dedicated to the related work. The ready-queue locking concept is described in detail in Section IV. Afterwards in Section V the schedulability test is provided for the scheduling policy. As a final theoretical contribution the ready-queue locking mechanism is integrated with preemption threshold, this is the subject of discussion of Section VI. Experimental results on uniformly generated tasksets are provided in the evaluation section VIII as a means to attest the performance of the proposed scheduling policies in this work. 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 priorities are assigned in a deadline monotonic fashion. The task τ_1 holds the highest priority and τ_n the lowest. The set represented by $hp(i)$ denotes the set of indexes of the tasks of higher priority than τ_i, which may be defined as $hp(i) = \{1, \ldots, i-1\}$. The set Γ_i contains the tasks with priority higher than τ_i. Each task is characterized by the four-tuple $\langle C_i, D_i, T_i, \mathrm{RQL}_i \rangle$. 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. $D_i \leq T_i$). The last task parameter (RQL_i) states the instant in time, relative to a job release, at which the job of task τ_i locks the ready queue if it still has pending workload. While the ready queue is locked job releases are not inserted into the ready queue. Fully preemptive and floating non-preemptive fixed priority scheduling policies are considered in this work. In the fully preemptive at every time the job with highest priority in the ready queue is executing in the processor, in the later model the preemption from the highest priority task in the ready queue is deferred for the maximum allowed time that still guarantees the task's correct temporal behaviour.

III. Related Work

A method for interference limitation was proposed by Express Logic [3] termed preemption threshold. In this work a task τ_j may only preempt another task τ_i if τ_j's priority is higher than that τ_i's preemption threshold. Wang and Saksena provided an optimal priority assignment for preemption-threshold scheduling policy [4]. 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. The preemption threshold values presented in this paper are the ones which are sufficient for ensuring schedulability. Later the work was extended [5] by the same authors to assign the preemption threshold to the highest possible priority value which maintains schedulability, thereby further reducing the number of preemptions.

A distinct model to limit the interference was described by Burns [6] (*fixed preemption points*). Keskin et al. discuss the theory of deferred preemption schedulability [7]. 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 job instance of 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.

The *fixed preemption points* methodology is exploited by Bertogna et al. [2] leading to a significant increase on the schedulability of the tasksets for fixed task-priority scheduling. This work has the limitation of only being suited for fixed preemptive regions.

The mechanism of preemption deferral has a number of advantages as has been pointed out in several works [8], [9], [2]. These scheduling policies present a trade-off between the extremes of non-preemptive and fully preemptive scheduling fixed task priority . Gang Yao et al. provide a comparison of all the available methods described so far in literature [8].

Gang Yao et al. [9] also provides a way bound the size of the floating non-preemptive regions. This upper-bound

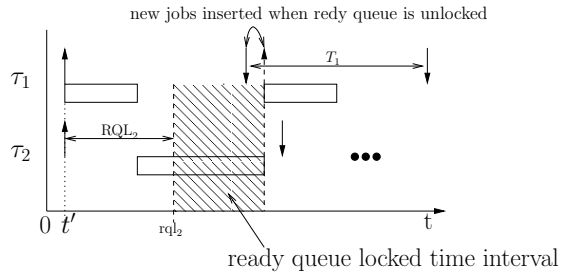


Figure 1. Ready-Q Locking Example.

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.

Gang Yao et al. devised a fixed priority scheduling method [10] 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 [9] because the last chunck of a task's execution is not taken into account during the analysis.

IV. Ready-Q locking concept

The ready-Q locking mechanism is introduced as a means to limit the amount of interference a task may suffer. It enables a job from a task to request that, after a certain time instant until its current workload completes, no other job is inserted into the ready queue. By preventing higher priority workload releases after a certain point in time the maximum interference a task may suffer is reduced. Each task τ_i has a ready-Q lock time instant defined (RQL_i). The RQL_i time instant is relative to release of the current job and $\mathrm{RQL}_i \leqslant D_i$. This translates into each job having a rql_i absolute time instant for which $\mathrm{rql}_i \leqslant d_i$, where d_i is the absolute deadline of the job. Since we consider a constrained deadline task model, at any time t there can only be at most one active job from each task in the system.

In Figure 1 an example is provided showing the benefits of the ready-queue locking mechanism. Consider the following taskset, composed of two tasks in an implicit deadline task model. The first task has $C_1 = 4$, $T_1 = 10$ and the second task has $C_2 = 7$, $T_2 = 12$. Assume a situation where these two tasks are synchronously released at a given time instant t', as is displayed in Figure 1. In fully preemptive fixed priority scheduling task τ_2 would suffer an interference of 6 time units from t' to $t' + T_2$, which would then leave only 6 time units for task τ_2 to execute its workload. since the ready queue was locked by τ_2 at $\mathrm{rql}_2 = t' + \mathrm{RQL}_2 = t' + 6$ it is then only subject to 4 time units of interference. Hence task τ_2 is able to complete its workload before the deadline, consequently releasing the lock on the ready queue.

A. Ready-Q Lock Implementation Considerations

The scheduler will manage a list of rql_i time instants for all active jobs (i.e. all the jobs in the ready queue at any time instant) from now onwards referred to as rlist. The list has at most $n - 1$ valid entries at any time. Each element in the rlist is composed by a time instant and the task to which it belongs to.

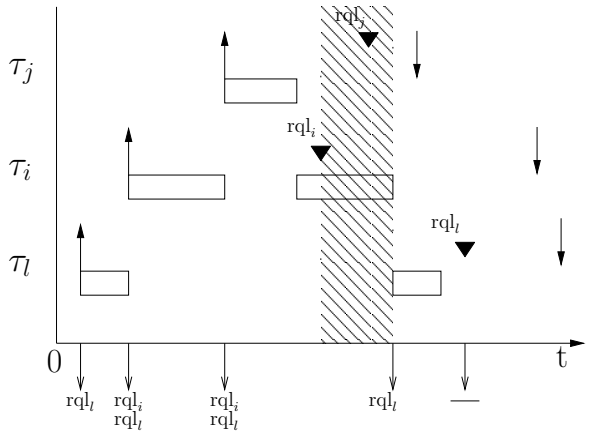


Figure 2. rlist List Evolution With Time

In Figure 2 a depiction of the rlist evolution with time is shown. At each relevant time instant an arrow points to the current rlist data structure. The solid black triangles represent rql_i time instants relative to each job depicted in the figure.

Algorithm 1: Ready Queue Locking Management

on event= release job from task τ_i:
$\mathrm{rql}_i \leftarrow \mathrm{release} + \mathrm{RQL}_i$
if *QueueIsLocked* **then**
  $\tau_l \leftarrow$ GetTaskLockingReadyQ()
  append job from τ_i to τ_l list of blocked tasks

on event= first dispatch of job from task τ_i:
if $\mathrm{rql}_i < \mathrm{rlist}[0]$ **then**
  $\mathrm{rlist.push}(\mathrm{rql}_i)$

on event= t $==$ rlist[0]:
assign the lock of the ready queue to the task which set $\mathrm{rlist}[0]$
$\mathrm{rlist.pop}()$

on event= terminate execution of τ_i job:
insert all tasks blocked by τ_i into the ready queue

In Algorithm 1 a pseudo-code description of the ready-Q locking management mechanism is presented. The mechanism is described as a set of callback procedures for the events of interest. When a job is dispatched for the first time (i.e. no workload was executed yet) the scheduler will get the rql_i from the task control block and evaluate its insertion into the rlist. In this situation, it holds true that the job being dispatched is the highest priority job in the ready queue. It also holds true that there can only be valid entries in the rlist belonging to jobs of lower or equal priority than the one being dispatched. If this would not hold then some higher priority jobs would be in the ready queue which implies that the current job could not be dispatched at this time. As a

consequence of this observation, we state that if two tasks τ_i and τ_j have jobs in the ready queue at some time t and $i > j$ then the response time of the job from τ_j will be smaller than the one from τ_i.

At time of first dispatch of a job from task τ_i the scheduler compares rql_i with the value on the top of the list. Two situations may then be observed.

1) rql_i is smaller than the value on top of the list, which leads to the insertion of rql_i in the rlist as the top element
2) rql_i is greater or equal than the value on top of the list, which leads to rql_i being discarded.

In the situation 1 there exist no job that will lock the ready queue before rql_i and since the job from task τ_i may need to lock the queue in order to complete its workload the value rql_i has to be considered. In the later situation (2) there exists a job from a lower priority task τ_l which will lock the ready queue before the job from task τ_i requires it. If by the time rql_i the job from τ_i still has pending workload, then the job from τ_l has pending workload as well. Which means that at time rql_i the ready queue is locked by the job from τ_l ($\mathrm{rql}_i > \mathrm{rql}_l$). The job from τ_i will then proceed to complete its workload without requiring to lock the ready queue since a lower priority task conducted that procedure on its behalf. The ready queue remains locked until the job from τ_l finishes its execution.

At any time instant t, if rlist is not empty, there exists a timer which will fire at time $t' = \mathrm{rlist}[0]$, the time instant at the top of rlist. When the timer fires, all the higher priority releases that occur after t' and before $\mathrm{rlist}[1]$ (if such value exists) are inserted into the ready queue once the job from the task that sets $\mathrm{rlist}[0]$ finishes its workload. In the timer event handler the head of rlist is popped. A task locks the ready queue between the time instant t' (at which a timer set by it fires) and a time instant t'' when either it completes execution or the timer set by some other task fires. Any job released in the interval $[t', t'']$ is inserted into a separate buffer which stores a pointer to all the tasks which are currently blocked. Once task τ_i finishes its workload the scheduler will check whether there were tasks blocked by task τ_i, in case there were, these are then moved from the separate queue into the ready queue with their correct priority.

In an extreme case RQL_i could be set to 0. This would constitute effectively non-preemptive scheduling, where once jobs from τ_i start to execute they would not suffer any further interference. This could jeopardize the timing properties of tasks with priority greater than i. The value of RQL_i has then to be decided upon considering the higher priority workload timing guarantees (i.e. no higher priority task can miss a deadline due to a lower priority one). These considerations are developed in the following sections.

When the ready queue is locked, the tasks released in the meantime are inserted into a separate circular buffer. The task τ_i that currently holds the ready queue lock has a pointer for the first task to be released while it held the lock, and another pointer to the last task to be released while it held the lock. Once task τ_i terminates the tasks in the circular buffer between the start and the end pointer are inserted into the ready queue. Notice that the complexity of operating this mechanism is reduced in comparison to EDF.

It is important to stress that all operations on the rlist have $O(1)$ complexity. A ready-queue lock is only enforced at the release of some higher priority task, and a new element can only be added to the rlist at the time of the first dispatch of a job. In the same manner an element in rlist is only removed when the job responsible for its insertion terminates.

V. Maximum Interference Computation

In this section the computation of the maximum interference a job might suffer in a ready queue locking framework is the subject of study. This computation is carried out on the maximum busy period of the level-i priority for task τ_i. The worst-case level-h busy period value represents the biggest amount of time workload from tasks with priority higher or equal than τ_i may execute continuously, assuming that all the level-h priority tasks are blocked by the maximum allowed time B. The largest of the level-i busy period occurs when all tasks in Γ_i are release synchronously with τ_i and are blocked by the longest time possible B [7].

The worst-case level-h busy period is defined as:

$$L_i(B) \stackrel{\text{def}}{=} \min\{t | t - (B + \mathrm{rbf}(\Gamma_i \cup \tau_i, t)) = 0\} \quad (1)$$

where

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

Let us assume that each task locks the ready queue at time RQL_i. A bound on the higher priority interference is required. This bound consists on all the higher priority releases, prior to the release of a job from task τ_i, which have not yet completed at the release time instant and all of the higher priority releases that occur since the release of τ_i and RQL_i.

Theorem 1. *The maximum interference any job from task τ_i is subject to is generated in the level-i busy period where the first job of task τ_i in the busy period is released ϕ time units after a synchronous release of all higher priority tasks, where $0 \leqslant \phi \leqslant L_{i-1}(Q_i)$, where Q_i denotes the maximum blocking time that the tasks in the set $\Gamma_i \cup \tau_i$ admit.*

Proof: Assume that a synchronous release of all higher priority tasks occurs at time $t = 0$ and that a release from τ_i occurs at $t = 0$ as well. In this scenario the interference the current job from task τ_i can suffer is I_0. Let us assume now that $\phi \neq 0$. For this case the interference the job from task τ_i is $I_0 - \phi$, provided that the number

of higher priority releases in the interval $[0, \mathrm{RQL}_i]$ is the same as for the interval $[0, \mathrm{RQL}_i + \phi]$. In a situation where the number of higher priority releases in the interval $[0, \mathrm{RQL}_i + \phi]$ is greater than the count from $[0, \mathrm{RQL}_i]$, then the interference suffered by the job of task τ_i is $I_0 - \phi + W$, where W is the additional workload released in the interval $[\mathrm{RQL}_i, \mathrm{RQL}_i + \phi]$ which will interfere with τ_i until D_i. The ϕ for which the interference is greater constitutes the worst-case scenario for a job from task τ_i.

After $L_{i-1}(Q_i)$ time units have elapsed since the previous synchronous release of all higher priority tasks has elapsed, mandatorily an idle time has occurred in the processor. If for some $\phi > L_{i-1}(Q_i)$ a situation of bigger amount of higher priority workload is generated then the critical scenario would not start with a synchronous release of all higher priority workload, which would contradict the first part of the proof. Since higher priority tasks can not endure a larger blocking time than Q_i then the same value is at most the maximum admissible blocking time for tasks of priority higher or equal to i. Hence the Theorem is proven. ■

Definition 1. *job frame: The time interval between a job release and its deadline is termed job frame.*

In order to find the maximum amount of interference one has then to find the correct higher priority synchronous release offset ϕ. The value ϕ is extracted by looking at the frame considering all the possible ϕ values in the interval $[0, L_{i-1}(Q_i)]$

Each frame of the task τ_i, in the busy period which starts with a synchronous release of the tasks in the level $i-1$ priority level, has to be checked for its temporal behaviour. The deadline is met in q^{th} frame if the slack in the frame is greater or equal than 0.

Let us assume define the following variable in order to ease the discussion:

$$r_i^q(\phi) = (q-1) \times T_i + \phi \tag{3}$$

The variable $r_i^q(\phi)$ encodes the value of the release of the q^{th} job from task τ_i relative to an absolute time instant, with a given ϕ offset.

All jobs in the level-i busy period have to be checked for deadline violations. In order to compute the number of jobs in the busy period, a maximum possible blocking for the level-i has to be considered. The level-i busy period can never be blocked by more than Q_i time units, since the tasks of priority greater than τ_i could otherwise suffer deadline misses. On the other hand τ_i can never be blocked by more than $D_i - C_i$, hence an upper bound on the number of jobs from τ_i in the busy period can be obtained considering $\min(Q_i, D_i - C_i)$ as the workload blocking the level-i busy period.

$$K = \left\lceil \frac{L_i(\min(Q_i, D_i - C_i))}{T_i} \right\rceil \tag{4}$$

The slack in each frame q may be expressed by:

$$\beta_i^q = \min_{\phi}\{\max(\max_{t \in \left[r_i^{q-1}(\phi), r_i^{q-1}(\phi) + \mathrm{RQL}_i\right]}\{t - (\mathrm{rbf}(\Gamma_i, t) + q \times C_i)\},$$
$$(r_i^{q-1}(\phi) + D_i) - (\mathrm{rbf}^*(\Gamma_i, r_i^{q-1}(\phi) + \mathrm{RQL}_i) + q \times C_i))\} \tag{5}$$

Where

$$\mathrm{rbf}^*(A, t) \stackrel{\mathrm{def}}{=} \sum_{\tau_j \in A} \left(\left\lfloor \frac{t}{T_j} \right\rfloor + 1 \right) \times C_j. \tag{6}$$

Notice that equations 2 and 6 only differ in value in multiples of T_j. The usage of both request bound function forms is tied to reducing equation display complexity. The Equation 5 is composed by two terms. In the first one the maximum idle time in the schedule is searched for in the interval between the release of a job from task τ_i and the ready queue locking time instant of the said job. In the second term the maximum idle time for the given frame is searched in the interval between the ready queue locking time instant of the job and its deadline, but since higher priority jobs released in this interval do not interfere with task τ_i it suffices to compute the idle time at the deadline of the job. For a given ϕ the maximum idle time is then the maximum value given by the two mentioned terms. Then for all the possible ϕ the minimum of the idle times is stored in β_i^q.

Considering all the frames the maximum amount of blocking that a job from task τ_i can endure is then defined as:

$$\beta_i = \min_q\{\beta_i^q\} \tag{7}$$

A simple depiction of the schedulability condition is provided in Figure 3. In this example the taskset is composed of three tasks. All tasks are implicit deadline tasks with parameters (T_i, C_i). Task τ_1 has $(5, 1)$, and τ_2, τ_3 have $(7, 2)$ and $(16, 4)$ respectively. It is implicitly assumed that $\mathrm{RQL}_i = D_i$ to simplify the visualization. The subject of schedulability analysis is in this case τ_3. Two frames from τ_3 are present in the figure. One can observe that $\beta_3^1 = 4$ and $\beta_3^2 = 8$. Since both frames present a non-negative slack value, then the τ_3 is deemed schedulable. Note that the maximum blocking time admissible for τ_3 is then 4 time units. This then implies that the level-3 busy period terminates at time instant 24, hence no more frames from task τ_3 require to be checked.

The maximum admissible blocking time for task τ_i is denoted by β_i. The task τ_i is schedulable if $\beta_i > 0$, when task τ_i is the task with lowest priority in the system.

In order for the task-set to be schedulable the RQL_i values for all the tasks have to be set so that the higher priority tasks

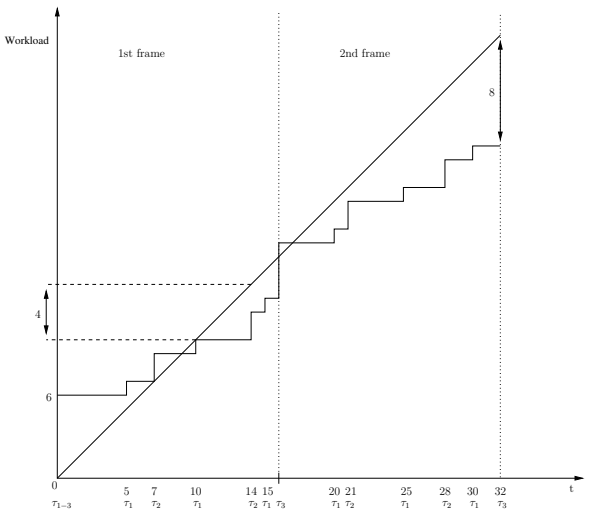


Figure 3. Schedulability Condition Depiction

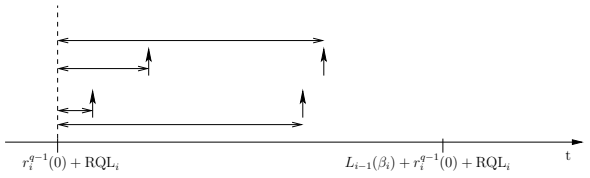


Figure 4. Set of Relevant Offsets for Frame q

temporal guarantees are still met. This matter is discussed in the next subsection.

For every q^{th} frame only a subset of the $\phi \in [0, L_{i-1}(Q_i)]$ needs to be checked. The higher priority workload increment for each frame occurs only at the boundary of the minimum inter arrival time of each higher priority task. Since only offsets up to the maximum busy-level period of the $i-1$ priority level needs to be checked as has been shown in Theorem 1, only the set of possible higher priority releases in the interval $[0, L_{i-1}(Q_i)]$ need to be checked. This set is defined as:

$$\Phi_i^q \stackrel{\text{def}}{=} \{0\} \cup \bigcup_{j \in hp(i)} \left(\left\{ \left\lceil \frac{r_i^{q-1}(0) + \mathrm{RQL}_i}{T_j} \right\rceil, \cdots, \left\lfloor \frac{L_{i-1}(\beta_i) + r_i^{q-1}(0) + \mathrm{RQL}_i}{T_j} \right\rfloor \right\} \times T_j - (r_i^{q-1}(0) + \mathrm{RQL}_i) \right) \quad (8)$$

In Figure 4 the set of relevant offsets for the given frame is graphically represented. The offsets are intuitively shown as the distance between the higher priority releases in the $\left[r_i^{q-1}(0) + \mathrm{RQL}_i, L_{i-1}(\beta_i) + r_i^{q-1}(0) + \mathrm{RQL}_i \right]$ interval. Of course $\phi = 0$ still has to be tested since the synchronous release of all the tasks in the $level - i$ priority band might still generate the critical interference scenario for task τ_i.

A. Ready-queue Locking Time Instant

The RQL_i value has to be such that τ_i has enough time to carry out its workload and the higher priority task are not blocked by more time than it is admissible.

Each task in the system may endure a maximum blocking time without endangering its timing guarantees [10] (i.e. all tasks may be blocked by a lower priority task up to a certain limit without missing any deadline)

The task with highest priority in the system can always sustain a maximum blocking time of $\beta_1 = D_1 - C_1$.

Let us assume that $\mathrm{RQL}_2 = D_2 - Q_2$. An hypothetical release from task τ_1 arriving at time instant RQL_2 would see its ready queue insertion delayed by task τ_2 by at most Q_2 time units. Since, by definition, $Q_2 \leqslant \beta_1$ this job from task τ_1 would still meet the deadline.

The concept may be generalized to n tasks by using the following relations:

$$Q_i \stackrel{\text{def}}{=} \begin{cases} 0 & \text{, if } i = 1 \\ \min_{j \in hp(i)}(\beta_j) & \text{, if } 1 < i \leqslant n \end{cases} \quad (9)$$

Equation 9 defines the maximum blocking time (Q_i) that all task of higher priority priority than i may endure without missing a deadline.

Assuming $\mathrm{RQL}_i = D_i - Q_i$ ensures then that, after locking the ready queue a job from task τ_i can only maintain it blocked by at most Q_i time units, hence not jeopardising higher priority task's temporal behaviour. Even if a synchronous release of all tasks in the $i-1$ priority level occurs after a ready-queue lock, at time instant rql_i then the queue will not be locked by more than Q_i.

In a scenario where $C_i < Q_i$ then $\mathrm{RQL}_i = D_i - C_i$. Since values are only inserted into the rlist at time of the first dispatch of a job, it might be the case that the first dispatch of the job occurs after rql_i, and hence the job might suffer more interference than it is admissible. By setting $\mathrm{RQL}_i = D_i - C_i$ it is ensured that the first dispatch of the job will always occur before rql_i.

For each task in the system the RQL_i is then set in the following manner:

$$\mathrm{RQL}_i = D_i - \min(Q_i, C_i) \quad (10)$$

Theorem 2. *The ready-queue locking scheduling policy dominates over fully preemptive fixed task priority.*

Proof: The maximum interference UI_i is smaller or equal to the interference the same τ_i task may endure without the ready-queue locking mechanism, hence all the

tasksets scheduled by fully preemptive fixed priority are schedulable with ready-queue locking. Since the UI_i may at times be smaller than the maximum interference in fully preemptive scheduling there exist tasksets schedulable by ready-queue locking which fail to be in fully preemptive scheduling. ■

It is worth noting that setting $\mathrm{RQL}_i = D_i - \min(Q_i, C_i)$ is not necessarily the optimal RQL_i time instant assignment. This is driven by the fact that an earlier locking will be possible in many situations, caused by a worst-case response time of a task which is shorter than RQL_i. Even more, the higher priority task τ_j which limits Q_i, may have a worst-case phasing such that an earlier RQL_i will not lead to a reduction of direct interference and thus have no negative effect on the higher priority task and a later than worst-case release would mean more progress for the locking task. However, deriving the optimal RQL_i is non-trivial and beyond the scope of this paper.

VI. Ready-q Locking with Preemption Threshold

The initial preemption-threshold assignment algorithm has been presented in [4]. The main drawback of the preemption-threshold assignment method is that it does not easily allow for a maximum blocking time per task computation. With this in mind the algorithm is rethought. As opposed to the solution proposed in [4] the taskset is parsed from the highest priority task to the lowest priority one.

Let us assume that the preemption threshold (π_i) of a task τ_i is $\pi_i = \min(j | C_i \leqslant \min_{k\in\{j+1,\cdots,i-1\}} \beta_k)$. Upon release a job from task τ_i is inserted into the ready queue (in case the ready queue is not locked) with priority i. At the time of its first dispatch onto the processor its priority is elevated to π_i. The set Γ_i^h denotes the set of tasks of higher priority than π_i, whereas the set Γ_i^l denotes the set of tasks with priority higher than i but lower than or equal to π_i. In order to ensure schedulability of the system, for each task, an initial upper-bound on the maximum blocking tolerance is provided. Initially $\beta_i = \min(D_i - C_i, Q_i)$, which is the maximum value that the blocking time could potentially take. The $level - i$ busy period is then parsed and checked for deadline misses.

Let us first define t_s^q as the worst-case time instant for the first dispatch of the q^{th} job from task τ_i. This time would be similar to computing the length of the $level - i$ busy period without considering the q^{th} job from task τ_i. where t_s^q is the worst-case instant in time when the q^{th} job of task τ_i starts to execute considering a blocking of β_i time units in the beginning of the busy period.

$$t_s^q = \min\{t | t - (\beta_i + rbf(\Gamma_i, t) + (q-1) \times C_i) \geqslant 0\} \tag{11}$$

Notice that if $t_s^q > r_i^q(0) + \mathrm{RQL}_i$ then the q^{th} job from task τ_i would miss a deadline since $D_i - \mathrm{RQL}_i \leqslant C_i$.

A deadline is missed in the q^{th} frame when there does not exist any idle time instant for the $level - i$ busy period in the interval $[t_s^q, r_i^q(\phi) + D_i]$. This condition may be written in the following form:

$$\nexists t \in [t_s^q, r_i^q(\phi) + \mathrm{RQL}_i] | t - (\beta_i + rbf(\Gamma_i^h, t_s^q) + rbf(\Gamma_i^l, t) + q \times C_i) \geqslant 0 \tag{12}$$

$$\wedge$$

$$D_i - (\beta_i + \mathrm{rbf}^*(\Gamma_i^h, t_s^q) + \mathrm{rbf}^*(\Gamma_i^l, r_i^q(\phi) + \mathrm{RQL}_i) + q \times C_i) \leqslant 0 \tag{13}$$

The condition expressed in 12 relates to the idle time occurrence in the interval between the release time of the q^{th} job of task τ_i and the instant in time at which it locks the ready queue. The second condition 13 relates to the search of idle time after the ready-queue is locked by task τ_i, in an interval where higher priority workload with higher priority than τ_i's preemption threshold do not interfere with the execution of τ_i.

If both conditions are met simultaneously, then for the given β_i a deadline may be missed in the q^{th} frame of task τ_i In this case, the β_i parameter has to be decreased.

$$\begin{aligned}\beta_i^- \stackrel{\mathrm{def}}{=} \min(&\min_{t\in A}\{t - (\beta_i + rbf(\Gamma_i, t) + (q-1) \times C_i)\},\\ &\min_{t\in[t_s^q, r_i^q(\phi)+\mathrm{RQL}_i]}\{t - (\beta_i + rbf(\Gamma_i^h, t_s^q) + rbf(\Gamma_i^l, t) + q \times C_i)\},\\ &D_i - (\beta_i + \mathrm{rbf}^*(\Gamma_i^h, t_s^q) + \mathrm{rbf}^*(\Gamma_i^l, r_i^q(\phi) + \mathrm{RQL}_i))\end{aligned} \tag{14}$$

where A is the set of time instants when higher priority releases occur in the time interval $[r_i^q(\phi), t_s^q]$:

$$A \stackrel{\mathrm{def}}{=} \{0\} \cup \bigcup_{j\in hp(i)} \left(\left\{ \left\lceil \frac{r_i^{q-1}(\phi)}{T_j} \right\rceil, \cdots, \left\lfloor \frac{t_s^q}{T_j} \right\rfloor \right\} \times T_j \right) \tag{15}$$

In Equation 14 there are three terms present. The first one relates to the t_s^q value, and both the second and the third relate to the idle time available in the q^{th} frame of task τ_i for the given ϕ parameter. In the second and third terms the minimum amount of β_i reduction required to compute all the workload in the frame is computed. In the first parameter the minimum β_i decrease which ensures that the t_s^q time instant occurs before one higher priority job is computed. The β decreased computed in the first term aims at decreasing the t_s^q such that the q^{th} job from τ_i starts earlier potentially suffering smaller interference from tasks of higher priority which is still lower than the τ_i preemption threshold. The minimum value between the three parameters is then chosen to be the value of β_i^- for the current algorithm iteration.

Finaly the current maximum blocking time allowed by task τ_i is obtained by:

$$\beta_i = \beta_i - \beta_i^- \tag{16}$$

If the deadline is met for the q^{th} frame with $\phi = 0$, using ready-q locking still requires all the possible offsets to be tested. As is the case for the ready-q locking mechanism only a subset of all possible ϕ values has to be tested, this subset Φ_i^q is defined in Equations 15.

When the following condition is met

$$\forall q \in \{0, \cdots, K\}, \forall \phi \in [0, L_{i-1}(\beta)], \exists t \in [r_i^q(0), r_i^q(0) + D_i] : \\ t - \beta_i + rbf(\Gamma_i^h, t_s^q) + rbf(\Gamma_i^l, t) + q \times C_i \geqslant 0 \tag{17}$$

then no deadlines are missed for task τ_i. Ensuring that the condition is met for all the tasks in the taskset will ensure the schedulability of the taskset.

Algorithm 2: Preemption Threshold Assignment with Ready-q Locking

Input : T
$\beta_1 = D_1 - C_1$
$\pi_1 = 1$
for $i = \{2, \cdots, n\}$ **do**
  $\pi_i = \min(j | C_i \leqslant \min_{k \in \{j+1, \cdots, i-1\}} \beta_k)$
  $Q_i = \min_{j \in hp(i)}\{\beta_j\}$
  $\mathrm{RQL}_i = D_i - \min(Q_i, C_i)$
  $\beta_i = \min(D_i - C_i, Q_i)$
  $K = \left\lceil \frac{L_i(\beta_i)}{T_j} \right\rceil$
  for $q \in \{1, \cdots, K\}$ **do**
    for $\phi \in \Phi_i^q$ **do**
      $t_s^q = \min\{t | t - (\beta_i + rbf(\Gamma_i^h, t) + rbf(\Gamma_i^l, t) + (q-1) \times C_i) \geqslant 0\}$
      while $\nexists t \in [t_s^q, r_i^q(\phi) + \mathrm{RQL}_i] | t - (\beta_i + rbf(\Gamma_i^h, t_s^q) + rbf(\Gamma_i^l, t) + q \times C_i) \geqslant 0 \wedge D_i - (\beta_i + \mathrm{rbf}^*(\Gamma_i^h, t_s^q) + \mathrm{rbf}^*(\Gamma_i^l, r_i^q(\phi) + \mathrm{RQL}_i) + q \times C_i) \leqslant 0$ **do**
        $\beta_i^- = \min(\min_{t \in A}\{t - (\beta_i + rbf(\Gamma_i, t) + (q-1) \times C_i)\}, \min_{t \in [t_s^q, r_i^q(\phi) + \mathrm{RQL}_i]}\{t - (\beta_i + rbf(\Gamma_i^h, t_s^q) + rbf(\Gamma_i^l, t) + q \times C_i)\}, D_i - (\beta_i + \mathrm{rbf}^*(\Gamma_i^h, t_s^q) + \mathrm{rbf}^*(\Gamma_i^l, r_i^q(\phi) + \mathrm{RQL}_i) + q \times C_i))$
        $\beta_i = \beta_i - \beta_i^-$
        if $\beta_i < 0$ **then**
          return *UNSCHED*
        $t_s^q = \min\{t | t - (\beta_i + rbf(\Gamma_i^h, t) + rbf(\Gamma_i^l, t) + (q-1) \times C_i) \geqslant 0\}$
return *SCHED*

The insertion of the rql_i value into the rlist data structure is considered at the first dispatch of a job, which coincides with the time instant of priority promotion of the job to π_i, hence the priority ordering between jobs with valid entries in the rlist is never changed after the insertion.

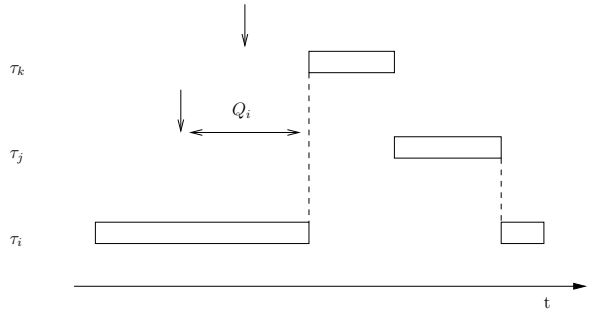


Figure 5. Floating Non-preemptive Region Scheduling Example

Notice that in a situation where $\mathrm{RQL}_i = D_i$ the provided schedulability test and preemption-threshold assignment technique is still valid for the regular [3] preemption-threshold mechanism.

Contrary to the method presented in [4] which assigns π_i the highest value that ensures schedulability (if such exists), Algorithm 2 assigns π_i the smallest possible value.

Notice that similarly to the Ready-q locking assigning $\mathrm{RQL}_i = D_i - \min(Q_i, C_i)$ is not optimal. Given the RQL_i value for each task, the schedulability test provided in Algorithm 2 is necessary and sufficient.

A. Floating Non-preemptive Regions

In a floating non-preemptive region scheduling, each task has the parameter Q_i defined. When a task τ_i is executing and τ_i is not the highest priority task in the ready queue, then it is said that a preemption deferral chain is occurring. A preemption deferral chain, as is shown in Figure 5 starts with a higher priority release, and lasts at most Q_i time units. At the end of a preemption deferral chain a preemption invariably happens. In Figure 5 task τ_i is executing when a job from τ_j is released, at some time in between a job from τ_k is released. The deferral chain ends exactly Q_i time units after the first higher priority release.

All the scheduling policies presented so far were created with the intention of enabling the floating non-preemptive regions usage. Effectively by extending the schedulability of fixed task priority as a consequence the Q_i values will be greater. Having bigger values for the non-preemptive regions is obviously advisable since this will inevitably allow for a reduction on the number of preemptions, and will enable less pessimism in the preemption delay computation [11]. In the simple ready queue locking, and in the ready queue locking with preemption threshold maximum blocking times admissible for all the tasks are computed. This information alone enables the usage of the floating non-preemptive regions scheduling.

VII. Preemption Upper Bounds

In this section a brief comparison between the preemption upper bound guarantees of the proposed solutions is pro-

vided. The value $WCRT_i$ denotes the worst-case response time of task τ_i. For fully preemptive

$$\sum_{j \in hp(i)} \left\lceil \frac{WCRT_i}{T_j} \right\rceil \tag{18}$$

The simple ready-queue locking scheduling policy ensures that preemptions may only occur in the time interval between release and the locking of the ready-queue. Hence with ready queue locking the worst-case number of preemptions will never be greater than in the fixed task priority scheduling.

$$\sum_{j \in hp(i)} \left\lceil \frac{\min(WCRT_i, \mathrm{RQL}_i)}{T_j} \right\rceil \tag{19}$$

The regular preemption threshold mechanism ensures that only tasks with higher priority than the preemption threshold priority may in fact preempt.

$$\sum_{j \in hp(\pi_i)} \left\lceil \frac{WCRT_i}{T_j} \right\rceil \tag{20}$$

For the preemption threshold with ready queue locking scheduling policy, the preemption upper-bound for each task is guaranteed to be smaller or equal than the value for any other policy described in this work. In the worst-case scenario the job from task τ_i would start execution immediately after release. In this situation assuming it executes for the worst-case execution time, suffering the worst possible interference it can be at most preempted during $\min(WCRT_i, \mathrm{RQL}_i)$ time units by the tasks with priority greater than π_i.

$$\sum_{j \in hp(\pi_i)} \left\lceil \frac{\min(WCRT_i, \mathrm{RQL}_i)}{T_j} \right\rceil \tag{21}$$

Obviously with floating non-preemptive regions the maximum number of preemptions each job would suffer is upper bounded by:

$$\left\lfloor \frac{C_i}{Q_i} \right\rfloor \tag{22}$$

As as been shown in [12], for small enough values of Q_i this bound is worse than the ones dictated by the higher priority releases. Hence for all the scheduling policies provided the upper bound on the number of preemptions is given by the $\min$(fully preemptive bound, $\left\lfloor \frac{C_i}{Q_i} \right\rfloor$).

VIII. Evaluation

In this section the proposed solutions are evaluated in terms of schedulability performance against fully preemptive fixed task-priority and regular preemption threshold.

Both proposed scheduling policies were evaluated with respect to schedulability. In each model all tasks are generated using the unbiased task-set generator method presented by Bini (UUniFast) [13]. Tasks are randomly generated for every utilization step in the set $\{0.8, 0.82, 0.85, 0.87, 0.93, 0.95, 0.97, 0.98\}$, their maximum execution requirements (C_i) were uniformly distributed in the interval $[20, 400]$. For every utilization step 1000 tasksets are trialled and checked whether the respective algorithm considers it schedulable. Task set sizes of 4, 8, and 16 tasks have been explored.

In the first situation the task-set behaves in a fully periodic manner with implicit deadlines ($D_i = T_i$). The results are depicted in Figures 6(a) to 6(c). Note, that the task set itself, might be sporadic, but the analysis does take only the minimal inter-arrival times and arbitrary phasings into account. 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 relative deadlines are constructed in the following manner $D_i = T_i - S$, where S is a random variable with uniform distribution in the interval $[0, 0.2 \times T_i]$. The results of these are put into juxtaposition with the implicit deadlines results in Figures 7(a) to 7(c).

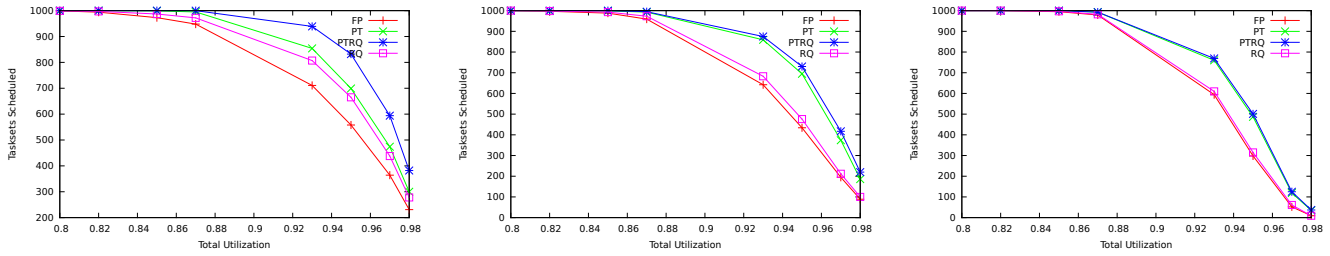
The data relative to regular fixed priority is tagged with FP. Preemption threshold is shown with tag PT. The simpler method of ready queue locking is addressed by RQ and the simple ready queue locking used together with preemption threshold is tagged with PTRQ.

A. Discussion

The pattern present in the results is clear. The simple ready-queue locking mechanism outperforms regular fixed priority as expected. However, it only rarely outperforms preemption threshold and that comparison deteriorates with increasing task set sizes and only gives it an overall gain for 4 tasks in the constrained deadline case. It is however noteworthy that there is no clear dominance relationship between PT and ready-queue locking, as some tasks sets are deemed schedulable with one, but not the other. This lack of dominance holds both for implicit and constrained deadlines models as well as for the different task set sizes investigated. The PTRQ solution performs always better than the simple preemption-threshold mechanism or simple reaqdy queue locking. Though again the benefits of PTRQ dilute with the increase of the taskset size.

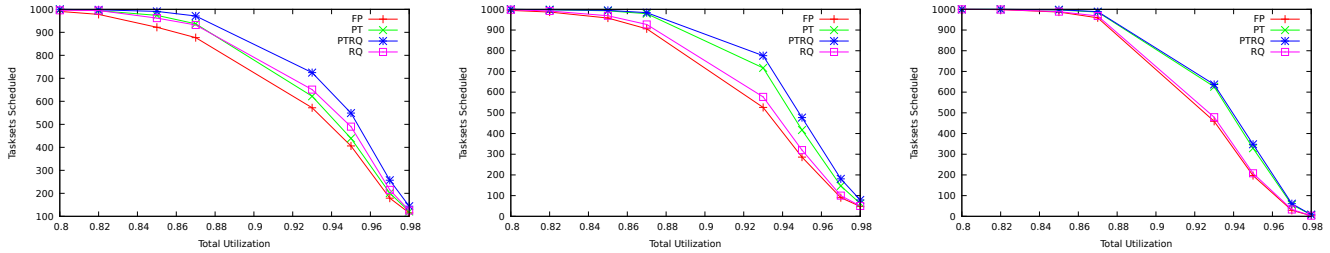
IX. Conclusions and Future Work

In this work several scheduling policies are proposed which aim to reduce the interference suffered by lower priority tasks in fixed task-priority scheduling. A new schedulability and preemption-threshold priority assignment algorithm is provided, which is much less complex than the one available in literature [4]. To the best of our knowledge, this is the first work which extends the schedulability of fixed task-priority systems and enables the usage of floating non-preemptive regions. Previous work either required fixed preemption points inserted into the tasks' code [10], or would not allow for the computation of the maximum



(a) Implicit Deadlines, 4 tasks (b) Implicit Deadlines, 8 tasks (c) Implicit Deadlines, 16 tasks

Figure 6. Simulation Results for the Implicit Task Model



(a) Constrained Deadlines, 4 tasks (b) Constrained Deadlines, 8 tasks (c) Constrained Deadlines, 16 tasks

Figure 7. Simulation Results for the Constrained Task Model

allowed blocking times of each task [4]. We have also shown that the preemption delay upper-bound per task is smaller in the case of the preemption threshold with ready queue locking scheduling policy. All the proposed mechanism maintain low complexity of operation while enabling a considerable increase on the schedulability of tasksets. As future work we intend to implement all the scheduling policies proposed in this work in a real-time kernel in order to assess its implementation overheads. Furthermore, we will devote more effort to identify an RQL instance closer to the optimal one to increase the performance of the RQL mechanisms.

REFERENCES

[1] S. Altmeyer, C. Maiza, and J. Reineke, "Resilience analysis: tightening the crpd bound for set-associative caches," in *LCTES 2010*.

[2] M. Bertogna, G. Buttazzo, M. Marinoni, G. Yao, F. Esposito, and M. Caccamo, "Preemption points placement for sporadic task sets," in *ECRTS 2010*.

[3] W. Lamie, "Preemption threshold," White paper, Express Logic, available online. [Online]. Available: http://rtos.com/articles/18833

[4] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *RTCSA 1999*.

[5] M. Saksena and Y. Wang, "Scalable real-time system design using preemption thresholds," in *RTSS 2000*.

[6] A. Burns, "Preemptive priority-based scheduling: an appropriate engineering approach," in *Advances in real-time systems*, S. H. Son, Ed. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1995.

[7] U. Keskin, R. Bril, and J. Lukkien, "Exact response-time analysis for fixed-priority preemption-threshold scheduling," in *ETFA 2010*.

[8] G. Yao, G. Buttazzo, and M. Bertogna, "Comparative evaluation of limited preemptive methods," in *ETFA 2010*.

[9] ——, "Bounding the maximum length of non-preemptive regions under fixed priority scheduling," *RTCSA 2009*.

[10] ——, "Feasibility analysis under fixed priority scheduling with limited preemptions," *Journal Real-Time Systems*, vol. 47, no. 3, 2011.

[11] J. Marinho, V. Nélis, S. M. Petters, and I. Puaut., "Preemption delay analysis for floating non-preemptive region scheduling," in *DATE 2012*.

[12] J. Marinho and S. M. Petters, "Job phasing aware preemption deferral," *EUC 2011*.

[13] E. Bini and G. Buttazzo, "Biasing effects in schedulability measures," in *ECRTS 2004*.