Technical Report

Extending Fixed Task-Priority Schedulability by Interference Limitation

José Marinho
Stefan M. Petters
Marko Bertogna

HURRAY-TR-120705
Version:
Date: 7/5/2012
Extending Fixed Task-Priority Schedulability by Interference Limitation

José Marinho, Stefan M. Petters, Marko Bertogna

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: jmsm@isep.ipp.pt, smp@isep.ipp.pt,
http://www.hurray.isep.ipp.pt

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.
Extending Fixed Task-Priority Schedulability by Interference Limitation

José Manuel Marinho  
CI TER-ISEP / INESC-TEC  
Porto, Portugal  
jmsm@isep.ipp.pt

Stefan M. Petters  
CI TER-ISEP / INESC-TEC  
Porto, Portugal  
smp@isep.ipp.pt

Marko Bertogna  
University of Modena  
Modena, Italy  
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.

Categories and Subject Descriptors
C.3 [Special-purpose and Application-based Systems]:  
[Real-time and embedded systems]

General Terms
Real-time Scheduling

Keywords
Dual Priority Scheduling, Non-preemptive Scheduling

1. 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 the cost of development and software maintenance, as well as that of later software updates. It also facilitates the operation of open systems which require the accommodation of workload changes at runtime.
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 when a higher priority job arrives when another task τi is executing. This non-preemptive region has a limited duration of Qi 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 model allows considerably more flexibility than the fixed non-preemptive regions one, the former cannot 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 model, 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 3 is dedicated to the related work. The ready-queue locking concept is described in detail in Section 4. In Section 5, 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 6. In Section 7, upper bounds are derived on the number of preemptions with the presented methods. Experimental results on uniformly generated task-sets are provided in the evaluation section 8 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.

2. 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 floating non-preemptive regions. The priorities are assigned in a deadline monotonic fashion. The task $\tau_1$ holds the highest priority and $\tau_n$ the lowest. The set represented by $\text{hp}(i)$ denotes the set of indexes of the tasks of higher priority than $\tau_i$, which may be defined as $\text{hp}(i) = \{1, \ldots, i - 1\}$. The set $\Gamma_i$ contains the tasks with priority higher than $\tau_i$. Each task is characterized by the four-tuple $(C_i, D_i, T_i, \text{RQL}_i)$. The parameter $C_i$ represents the worst-case execution time of each job from $\tau_i$, $D_i$ 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 of jobs (i.e. each task $\tau_i$ may release a potentially infinite sequence of jobs separated by at least $T_i$ time units) and constrained deadlines (i.e. $D_i \leq T_i$). The last task parameter ($\text{RQL}_i$) states the instant in time, relative to a job release, at which the job of task $\tau_i$ 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.

3. RELATED WORK

A method for interference limitation was proposed by Express Logic [6] termed preemption threshold. In this work a task $\tau_j$ may only preempt another task $\tau_i$ if $\tau_j$’s priority is higher than that $\tau_i$’s preemption threshold. Wang and Saksena provided an optimal priority assignment for preemption-threshold scheduling policy [10]. 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 [9] 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 [4] (fixed preemption points). Keskin et al. discuss the theory of deferred preemption schedulability [5]. The author deemed the available test [4] optimistic, arguing that the worst-case response time for a job of task $\tau_i$ is not necessarily given by the first job instance of a synchronous release situation, but that it may show up in a job of task $\tau_i$ 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 task-sets 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 [12, 11, 2]. These scheduling policies present a trade-off between the extremes of non-preemptive and fully preemptive scheduling fixed task priority. Yao et al. provide a comparison of all the available methods described so far in literature [12].
In [11], a way to bound the size of the floating non-preemptive regions is provided. This upper-bound is computed using the request bound function. It basically derives the amount of idle time ($\beta_i$) for the critical region of task $\tau_i$ in a synchronous release situation.

Yao et al. devised a fixed priority scheduling method [13] where a maximum bound on the length fixed non-preemptive regions is provided. In this situation the computed $\beta_i$'s are generally larger than in the previous work [11] because the last chunk of a task's execution is not taken into account during the analysis.

4. 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 $\tau_i$ has a ready-Q lock time instant defined (RQL$_i$). The RQL$_i$ time instant is relative to release of the current job and RQL$_i$ $\leq$ $D_i$. This translates into each job having a rql$_i$ absolute time instant for which rql$_i$ $\leq$ $d_i$, 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 task-set, 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 $\tau_2$ would suffer an interference of 6 time units from $t'$ to $t' + T_2$, which would then leave only 6 time units for task $\tau_2$ to execute its workload. Since the ready queue was locked by $\tau_2$ at rql$_2 = t' + \text{RQL}_2 = t' + 6$ it is then only subject to 4 time units of interference. Hence task $\tau_2$ is able to complete its workload before the deadline, consequently releasing the lock on the ready queue.

![Image of Ready-Q Locking Example](image1.png)

**Figure 1: Ready-Q Locking Example.**

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

![Image of Evolution of rql over Time](image2.png)

**Figure 2: Evolution of rlist over Time**

4.1 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.

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**

```plaintext
on event= release job from task $\tau_i$:
    rql$_i$ ← release + RQL$_i$
    if QueueIsLocked then
        $\tau_i$ ← GetTaskLockingReadyQ()
        append job from $\tau_i$ to $\tau_i$ list of blocked tasks
    on event= first dispatch of job from task $\tau_i$:
        if rql$_i$ $<$ rlist[0] then
            rlist.push(rql$_i$)
        on event= $t$ $==$ rlist[0]:
            assign the lock of the ready queue to the task which
            set rlist[0]
            rlist.pop()
        on event= terminate execution of $\tau_i$ job:
            insert all tasks blocked by $\tau_i$ 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 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 \( \tau_i \) and \( \tau_j \) have jobs in the ready queue at some time \( t \) and \( i > j \) then the response time of the job from \( \tau_j \) will be smaller than the one from \( \tau_i \).

At time of first dispatch of a job from task \( \tau_i \), the scheduler compares rql with the value on the top of the list. Two situations may then be observed.

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

In the situation 1 there exist no job that will lock the ready queue before rql, and since the job from task \( \tau_i \) may need to lock the queue in order to complete its workload the value rql has to be considered. In the later situation (2) there exists a job from a lower priority task \( \tau_j \) which will lock the ready queue before the job from task \( \tau_i \) requires it. If by the time rql the job from \( \tau_i \) still has pending workload, then the job from \( \tau_j \) has pending workload as well. Which means that at time rql the ready queue is locked by the job from \( \tau_j \) (rql > rql). The job from \( \tau_j \) 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 \( \tau_j \) finishes its execution.

At any time instant \( t \), if rlist is not empty, there exists a timer which will fire at time \( t' = \text{rlist}[0] \), the time instant at the top of rlist. When the timer fires, all the higher priority releases that occur after \( t' \) and before \( \text{rlist}[1] \) (if such value exists) are inserted into the ready queue once the job from the task that sets \( \text{rlist}[0] \) 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 \( \tau_j \) finishes its workload the scheduler will check whether there were tasks blocked by task \( \tau_j \), 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 can be set to 0. This would constitute effectively non-preemptive scheduling, where once jobs from \( \tau_i \) 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 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 \( \tau_i \) 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 \( \tau_i \) 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.

5. 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 \( \tau_i \). The worst-case level-h busy period value represents the biggest amount of time workload from tasks with priority higher or equal than \( \tau_i \) may execute continuously, assuming that all the level-h priority tasks are blocked by the maximum allowed time. The largest of the level-i busy period occurs when all tasks in \( \Gamma_i \) are release synchronously with \( \tau_i \) and are blocked by the longest time possible \( B \).

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

\[
L_i(B) \overset{\text{def}}{=} \min \{ t | t - (B + \text{rbf}(\Gamma_i \cup \tau_i, t)) = 0 \} \tag{1}
\]

where

\[
\text{rbf}(A, t) \overset{\text{def}}{=} \sum_{\tau_j \in A} \left[ \frac{C_j}{T_j} \right] \times C_j. \tag{2}
\]

Let us assume that each task locks the ready queue at time \( \text{RQL}_i \). 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 \( \tau_i \), which have not yet completed at the release time instant and all of the higher priority releases that occur since the release of \( \tau_i \) and \( \text{RQL}_i \).

**Theorem 1.** The maximum interference any job from task \( \tau_i \) is subject to is generated in the level-i busy period where the first job of task \( \tau_i \) in the busy period is released \( \phi \) time units after a synchronous release of all higher priority tasks, where \( 0 \leq \phi \leq L_{i-1}(Q_i) \), 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
\( \tau_i \) occurs at \( t = 0 \) as well. In this scenario the interference the current job from task \( \tau_i \) can suffer is \( I_0 \). Let us assume now that \( \phi \neq 0 \). For this case the interference the job from task \( \tau_i \) is \( I_0 - \phi \), provided that the number of higher priority releases in the interval \([0, RQ\text{L}_i]\) is the same as for the interval \([0, RQ\text{L}_i + \phi]\). In a situation where the number of higher priority releases in the interval \([0, RQ\text{L}_i + \phi]\) is greater than the count from \([0, RQ\text{L}_i]\), then the interference suffered by the job of task \( \tau_i \) is \( I_0 - \phi + W \), where \( W \) is the additional workload released in the interval \([RQ\text{L}_i, RQ\text{L}_i + \phi]\) which will interfere with \( \tau_i \) until \( D_i \). The \( \phi \) for which the interference is greater constitutes the worst-case scenario for a job from task \( \tau_i \).

After \( L_{i-1}(Q_i) \) time units have elapsed since the previous synchronous release of all higher priority tasks, 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 proved.

**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 \( \phi \). The value \( \phi \) is extracted by looking at the frame considering all the possible \( \phi \) values in the interval \([0, L_{i-1}(Q_i)]\).

Each frame of the task \( \tau_i \), 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 define the following variable in order to ease the discussion:

\[
q_i(\phi) = (q - 1) \times T_i + \phi
\]  

The variable \( q_i(\phi) \) encodes the value of the release of the \( q^{th} \) job from task \( \tau_i \) relative to an absolute time instant, with a given \( \phi \) 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 \( \tau_i \) could otherwise suffer deadline misses. On the other hand \( \tau_i \) can never be blocked by more than \( D_i - C_i \), hence an upper bound on the number of jobs from \( \tau_i \) in the busy period can be obtained considering \( \min(Q_i, D_i - C_i) \) as the workload blocking the level-\( i \) busy period.

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

![Figure 3: Schedulability Condition Depiction](image)

The slack in each frame \( q \) may be expressed by:

\[
\beta_i^q = \min_{\phi} \{ \max_{t \in \left[ 0, T_i(\phi) \right]} \left\{ t - (\text{rbf}(\Gamma_i, t) + q \times C_i) \right\} , \left( r_i^{q-1}(\phi) + D_i \right) - (\text{rbf}(\Gamma_i, r_i^{q-1}(\phi) + RQ\text{L}_i) + q \times C_i) \} \}
\]  

Where

\[
\text{rbf}^q(A, t) \overset{\text{def}}{=} \sum_{t_j \in A} \left( \left\lfloor \frac{t}{T_j} \right\rfloor + 1 \right) \times C_j.
\]

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 \( \tau_i \) 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. Since higher priority jobs released in this interval do not interfere with task \( \tau_i \), it suffices to compute the idle time at the deadline of the job. For a given \( \phi \), the maximum idle time is then the maximum value given by the two mentioned terms. Then, for all the possible \( \phi \) the minimum of the idle times is stored in \( \beta_i^q \).

Considering all the frames, the maximum amount of blocking that a job from task \( \tau_i \) can endure is then defined as:

\[
\beta_i = \min_{q} \{ \beta_i^q \}
\]

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 \(\tau_1\) has \((5, 1)\), and \(\tau_2, \tau_3\) have \((7, 2)\) and \((16, 4)\) respectively. It is implicitly assumed that \(\text{RQL}_{\tau_1} = D_1\) to simplify the visualization. The subject of schedulability analysis is in this case \(\tau_2\). Two frames from \(\tau_2\) are present in the figure. One can observe that \(\beta_1^2 = 4\) and \(\beta_2^2 = 8\). Since both frames present a non-negative slack value, then the \(\tau_2\) is deemed schedulable. Note that the maximum blocking time admissible for \(\tau_3\) 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 \(\tau_3\) require to be checked.

The maximum admissible blocking time for task \(\tau_1\) is denoted by \(\beta_1\). The task \(\tau_1\) is schedulable if \(\beta_1 > 0\), when task \(\tau_1\) is the task with lowest priority in the system.

In order for the task-set to be schedulable the \(\text{RQL}_i\) values for each task have to be set so that the higher priority tasks 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:

\[
\Phi_q^i \triangleq \{0\} \cup \bigcup_{j \in h_p(i)} \left\{ \left[ T_j \left( L_{i-1}(\beta_j) - r_{i-1}(0) + \text{RQL}_i \right) \right] \times \left[ \left[ \frac{r_{i-1}(0) + \text{RQL}_i}{T_j} \right] + \left[ \frac{L_{i-1}(\beta_j) - r_{i-1}(0) + \text{RQL}_i}{T_j} \right] \right] \right\}
\]

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 \(\left[ r_{i-1}(0) + \text{RQL}_i, L_{i-1}(\beta_i) + r_{i-1}(0) + \text{RQL}_i \right]\) 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 \(\tau_i\).

### 5.1 Ready-queue Locking Time Instant

The \(\text{RQL}_i\) value has to be such that \(\tau_i\) 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 [13] (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 \(D_1 - C_1\).

Let us assume that \(\text{RQL}_2 = D_2 - Q_2\). An hypothetical release from task \(\tau_1\) arriving at time instant \(\text{RQL}_2\) would see its ready queue insertion delayed by task \(\tau_2\) by at most \(Q_2\) time units. Since, by definition, \(Q_2 < D_1\) this job from task \(\tau_1\) would still meet the deadline.

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

\[
Q_i \triangleq \begin{cases} 
0, & \text{if } i = 1 \\
\min_{j \in h_p(i)} \beta_j, & \text{if } 1 < i \leq n 
\end{cases}
\]

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

Assuming \(\text{RQL}_i = D_i - Q_i\) ensures then that, after locking the ready queue, a job from task \(\tau_i\) 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 \(\text{rq}_{i-1}\) the queue will not be locked by more than \(Q_i\).

In a scenario where \(Q_i < D_i\) then \(\text{RQL}_i = D_i - C_i\). Since values are only inserted into the list at time of the first dispatch of a job, it might be the case that the first dispatch of the job occurs after \(\text{rq}_{i-1}\), and hence the job might suffer more interference than it is admissible. By setting \(\text{RQL}_i = D_i - C_i\) it is ensured that the first dispatch of the job will always occur before \(\text{rq}_{i-1}\).

For each task in the system, the \(\text{RQL}_i\) is then set in the following manner:

\[
\text{RQL}_i = D_i - \min(Q_i, C_i)
\]

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

**Proof.** The maximum interference \(U_{i-1}\) is smaller or equal to the interference the same task \(\tau_i\) may endure without the ready-queue locking mechanism. Hence all the task-sets scheduled by fully preemptive fixed priority are schedulable with ready-queue locking. Since the \(U_{i-1}\) may at times be smaller than the maximum interference in fully preemptive scheduling there exist task-sets schedulable by ready-queue locking which fail to be in fully preemptive scheduling.

It is worth noting that setting \(\text{RQL}_i = D_i - \min(Q_i, C_i)\) is not necessarily the optimal \(\text{RQL}_i\) 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 \(\text{RQL}_i\). Even more, the higher priority task \(\tau_i\) which limits \(Q_i\) may have a worst-case phasing such that an earlier \(\text{RQL}_i\) will not lead to a
6. READY-Q LOCKING WITH PREEMPTION THRESHOLD

The initial preemption-threshold assignment algorithm has been presented in [10]. 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 [10] the taskset is parsed from the highest priority task to the lowest priority one.

Let us assume that the preemption threshold \( \pi_t \) of a task \( \tau_t \) is \( \pi_t = \min(j \mid C_j \leq \min(k \mid j + 1, \ldots, i - 1) \beta_i) \). Upon release a job from task \( \tau_t \) 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 \( \tau_t \). The set \( \Gamma_i^k \) denotes the set of tasks of higher priority than \( \tau_t \), whereas the set \( \Gamma_i^l \) denotes the set of tasks with priority higher than \( i \) but lower than or equal to \( \pi_t \). 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 \( \tau_t \) busy period is then parsed and checked for deadline misses.

Let us first define \( \ell_t^p \) as the worst-case time instant for the first dispatch of the \( q^{th} \) job from task \( \tau_t \). This time would be similar to computing the length of the \( \tau_t \) busy period without considering the \( q^{th} \) job from task \( \tau_t \), where \( \ell_t^p \) is the worst-case instant in time when the \( q^{th} \) job of task \( \tau_t \) starts to execute considering a blocking of \( \beta_i \) time units in the beginning of the busy period.

\[
\ell_t^p = \min(t - (\beta_i + rbf(T_1, t) + (q - 1) \times C_i) \geq 0) \quad (11)
\]

Notice that if \( \ell_t^p > r_f^q(0) + RQL \) then the \( q^{th} \) job from task \( \tau_t \) would miss a deadline since \( D_t - RQL \leq C_i \).

A deadline is missed in the \( q^{th} \) frame when there does not exist any idle time instant for the \( \tau_t \) busy period in the interval \( [r_{1}^{q}(\phi) + D_i] \). This condition may be written in the following form:

\[
\exists t \in [r_{1}^{q}(\phi) + RQL] \mid t - (\beta_i + rbf(T_1, \ell_t^p) + rbf(T_1, t) + q \times C_i) \geq 0 \quad (12)
\]

\[
D_t - (\beta_i + rbf(T_1, \ell_t^p) + rbf(T_1, r_{1}^{q}(\phi) + RQL)) + q \times C_i \leq 0 \quad (13)
\]

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 \( \tau_t \) 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 \( \tau_t \), in an interval where higher priority workload with higher priority than \( \tau_t \)'s preemption threshold do not interfere with the execution of \( \tau_t \).

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

\[
\beta_t^\prime \equiv \min\{\min\{t - (\beta_t + rbf(T_1, t) + (q - 1) \times C_i)\},
\]

\[
\min\{t - (\beta_t + rbf(T_1, \ell_t^p) + rbf(T_1, t) + q \times C_i)\},
\]

\[
D_t - (\beta_t + rbf(T_1, \ell_t^p) + rbf(T_1, r_{1}^{q}(\phi) + RQL)) + q \times C_i \}
\]

\[
(14)
\]

where \( A \) is the set of time instants when higher priority releases occur in the time interval \( [r_{1}^{q}(\phi), \ell_t^p] \):

\[
A \equiv \{0\} \cup \bigcup_{j \in \epsilon_p(i)} \left\{ \left( \frac{r_{1}^{q-1}(\phi)}{T_j}, \ldots, \frac{\ell_t^p}{T_j} \right) \times T_j \right\}
\]

In Equation 14 there are three terms present. The first one relates to the \( t^0 \) value, and both the second and the third relate to the idle time available in the \( q^{th} \) frame of task \( \tau_t \) for the given \( \phi \) parameter. In the second and third terms the minimum amount of \( \beta_t \) reduction required to compute all the workload in the frame is computed. In the first parameter the minimum \( \beta_t \) decrease which ensures that the \( t^0 \) time instant occurs before one higher priority job is computed. The \( \beta_t \) decreased computed in the first term aims at decreasing the \( t^0 \) such that the \( q^{th} \) job from \( \tau_t \) starts earlier potentially suffering smaller interference from tasks of higher priority which is still lower than the \( \tau_t \) preemption threshold. The minimum value between the three parameters is then chosen to be the value of \( \beta_t^\prime \) for the current algorithm iteration.

Finally the current maximum blocking time allowed by task \( \tau_t \) is obtained by:

\[
\beta_t = \beta_t - \beta_t^\prime \quad (16)
\]

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 \( \phi \) values has to be tested, this subset \( \Phi_q \) is defined in Equations 15.

When the following condition is met

\[
\forall q \in \{0, \ldots, K\}, \forall \phi \in [0, L_i - (\beta_t)], \exists t \in [r_{1}^{q}(0), r_{1}^{q}(0) + D_i] : \\
t - (\beta_t + rbf(T_1, \ell_t^p) + rbf(T_1, t) + q \times C_i) \geq 0
\]

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

The insertion of the rql 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 \( \tau_t \), hence the priority ordering between jobs with valid entries in the rlist is never changed after the insertion.
Algorithm 2: Preemption Threshold Assignment with Ready-q Locking

Input : $T$

$\beta_1 = D_1 - C_1$

$\pi_1 = 1$

for $i = 2, \ldots , n$ do

$\pi_i = \min(j | C_j \leq \min_{k \in \{j+1, \ldots , n-1\}} \beta_k)$

$Q_i = \min_{j \in hpt(i)} \{\beta_j\}$

$RQL_i = D_i - \min(Q_i, C_i)$

$\beta_i = \min(D_i - C_i, Q_i)$

$K = \left\lceil \frac{L_{i}(\beta_i)}{RQL_i} \right\rceil$

for $q \in \{1, \ldots , K\}$ do

for $\phi \in \Phi_i^q$ do

$\mathbf{t}_q^i = \min\{t | (\beta_i + rbf(\Gamma_i^h, t) + rbf(\Gamma_i^h, t) + (q - 1) \times C_i) \geq 0 \}$

while $\mathbf{t}_q^i \in [t_q^i, r_q^i(\phi) + RQL_i][t - (\beta_i + rbf(\Gamma_i^h, t_q^i) + rbf(\Gamma_i^h, t_q^i) + q \times C_i) \geq 0 \land D_i - (\beta_i + rbf(\Gamma_i^h, t_q^i) + rbf(\Gamma_i^h, t_q^i) + rbf(\Gamma_i^h, r_q^i(\phi) + RQL_i) + q \times C_i) \leq 0)$ do

$\beta_i = \min\{\min_{\mathbf{t}_q^i \in [t_q^i, r_q^i(\phi) + RQL_i]} \{t - (\beta_i + rbf(\Gamma_i^h, t_q^i) + rbf(\Gamma_i^h, t_q^i) + q \times C_i) \}$

if $\beta_i < 0$ then

$\mathbf{return UNSCHED}$

$\mathbf{t}_q^i = \min\{t | (\beta_i + rbf(\Gamma_i^h, t) + rbf(\Gamma_i^h, t) + (q - 1) \times C_i) \geq 0 \}$

return $\mathbf{SCHED}$

Notice that in a situation where $RQL_i = D_i$ the provided schedulability test and preemption-threshold assignment technique is still valid for the regular [6] preemption-threshold mechanism.

Contrary to the method presented in [10] which assigns $\pi_i$ the highest value that ensures schedulability (if such exists), Algorithm 2 assigns $\pi_i$ the smallest possible value.

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

6.1 Floating Non-preemptive Regions

In a floating non-preemptive region scheduling, each task has the parameter $Q_i$ defined. When a task $\tau_i$ is executing and $\pi_i$ is 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 $\tau_i$ is executing when a job from $\tau_j$ is released, at some time in between a job from $\tau_i$ 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 [7]. 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.

7. PREEMPTION UPPER BOUNDS

In this section a brief comparison between the preemption upper bound guarantees of the proposed solutions is provided. The value $WCRT_i$ denotes the worst-case response time of task $\tau_i$. For fully preemptive

$$\sum_{j \in hpt(i)} \left[ \frac{WCRT_j}{T_j} \right]$$ (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.

$$\sum_{j \in hpt(i)} \left[ \frac{\min(WCRT_j, RQL_i)}{T_j} \right]$$ (19)

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

$$\sum_{j \in hpt(\pi_i)} \left[ \frac{WCRT_j}{T_j} \right]$$ (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 \( \tau_i \) 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 \( \min(WC_{RT}, RQ_{i}) \) time units by the tasks with priority greater than \( \pi_i \).

\[
\sum_{j \in E_{pt}(\tau_i)} \left\lfloor \frac{\min(WC_{RT}, RQ_{j})}{T_j} \right\rfloor
\]

(21)

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

\[
\left\lfloor \frac{C_i}{Q_i} \right\rfloor
\]

(22)

As as been shown in [8], 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 \( \min(\text{fully preemptive bound}, \left\lfloor \frac{C_i}{Q_i} \right\rfloor) \).

8. 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) [3]. Tasks are generated for every utilization step in the set \( \{0.8, 0.82, 0.85, 0.87, 0.93, 0.95, 0.97, 0.98\} \) randomly. Their maximum execution requirements \( (C_i) \) were uniformly distributed in the interval \([20, 400]\). For every utilization step 1000 task-sets 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.

8.1 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 ready queue locking. Though again the benefits of PTRQ dilute with the increase of the taskset size.

9. 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 [10]. 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 [13], or would not allow for the computation of the maximum allowed blocking times of each task [10]. 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 task-sets. 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.

Acknowledgements

This work was partially supported by national funds through the FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme ‘Thematic Factors of Competitiveness’), within REPOMUC project, ref. FCOMP-01-0124-FEDER-015050, by FCT and the EU by ARTEMIS JU funding, within RECOMP project, ref. ARTEMIS/0202/2009, JU grant nr. 100202 and by FCT and the ESF (European Social Fund) through POPS (Portuguese Human Potential Operational Program), under PhD grant SFRH/BD/81085/2011.
10. REFERENCES


