Multiprocessor Fixed Priority Scheduling with Limited Preemptions

Abhilash Thekkilakattil[1], Robert I. Davis[2], Radu Dobrin[1], Sasikumar Punnekkat[3], and Marko Bertogna[4]

[1]Mälardalen Real-Time Research Center, Mälardalen University, Sweden

[2]Real-Time Systems Research Group, Department of Computer Science, University of York, UK

[3]Department of Computer Science and Information Systems, Birla Institute of Technology and Science, India

[4]University of Modena, Italy

ABSTRACT

Challenges associated with allowing preemptions and migrations are compounded in multicore systems, particularly under global scheduling policies, because of the potentially high overheads. For example, multiple levels of cache greatly increase preemption and migration related overheads as well as the difficulty involved in accurately accounting for them, leading to substantially inflated worst-case execution times (WCETs). Preemption and migration related overheads can be significantly reduced, both in number and in size, by using fixed preemption points in the tasks' code; thus dividing each task into a series of non-preemptive regions (NPRs). This leads to an additional consideration in the scheduling policy. When a high priority task is released and all of the processors are executing non-preemptive regions of lower priority tasks, then there is a choice to be made in terms of how to manage the next preemption. With an eager *approach the first lower priority task to reach a preemption point is preempted even if it is not the lowest priority running task. Alternatively, with a* lazy *approach, preemption is delayed until the* lowest *priority currently running task reaches its next preemption point.*

In this paper, we show that under global fixed priority scheduling with eager preemptions each task suffers from at most a single priority inversion each time it resumes execution. Building on this observation, we derive a new response time based schedulability test for tasks with fixed preemption points. Experimental evaluations show that global fixed priority scheduling with eager preemptions is significantly more effective than with lazy preemption using link based scheduling in terms of task set schedulability.

1. INTRODUCTION

RTNS 2015, Lile, France

Multicore platforms have been developed to circumvent the fact that performance gains can no longer be achieved via further increases in processor speed, due to power and thermal constraints [3]. Consequently, there is a revived interest among researchers and practitioners, particularly in the field of real-time embedded systems, to leverage on this ability of multicore systems to provide higher performance. In particular, the research focus has been on the challenges associated with such a hardware-software ecosystem, for example, the associated unpredictabilities that may compromise timeliness. A compelling challenge involves minimizing preemption and migration related overheads while enabling preemptions on lower priority tasks to aid the timely completion of both higher and lower priority ones.

Preemption and migration related overheads include context switch costs, cache related preemption and migration delays (CPMD), pipeline delays, and increased bus contention costs [4] [9], all of which significantly increases pessimism in task execution times. Consequently, fully preemptive scheduling can lead to prohibitively high preemption related overheads which degrade schedulability and can potentially cause deadline failure if not accurately accounted for. Two main reasons for high preemption and migration related overheads can be identified in this case. First, there are a large number of possible preemptions and migrations (resulting from the preemptions), and second these preemptions and migrations can occur at points in the code where the associated overhead is very high, for example where there are many (useful) cache blocks. Bastoni *et al.* [2] showed that the overheads due to preemptions and migrations are similar.

Fully non-preemptive scheduling offers an alternative that avoids preemption related overheads, at the cost of introducing significant blocking effects. While fixed priority preemptive and non-preemptive scheduling are *incomparable* for single processor systems [12], and also in the case of global scheduling for multiprocessor systems [18], non-preemptive scheduling suffers from what is known as the *long task problem*. This is where one task has a Worst Case Execution Time (WCET) that is longer than the deadline of another task, rendering the system unschedulable on a single processor. This problem is somewhat alleviated on multiprocessor systems, since long tasks need to be executing on all processors in order to

cause a deadline miss [18]; however, it is still an issue.

Limited preemption scheduling has been proposed as an alternative to the fully preemptive and fully non-preemptive paradigms in order to reduce preemption and migration related overheads, while also constraining the amount of blocking and thus improving schedulability. In limited preemption scheduling, preemptions and migrations may be restricted using various mechanisms such as scheduler enforced floating non-preemptive regions, or fixed preemption points inserted into the program code. (See [10] for a survey of the various methods). Fixed preemption points, in addition to reducing the number of preemptions and migrations, have the advantage of enabling accurate estimation and accounting for preemption and migration related overheads. Further, preemption point placement can be optimized to reduce the cost of each individual preemption and the total preemption related overheads incurred within the response time of a task [4], [6], [21]. Moreover, in multiprocessors, by reducing the number of preemption points, the number of potential migrations can also be reduced.

In this paper, we consider the problem of global fixed priority scheduling of real-time tasks with fixed preemption points on a multiprocessor. Here, there is an interesting choice to be made in terms of how to manage preemptions following the release of high priority tasks. Two types of approach can be identified: With an *eager* approach, preemption occurs as soon as possible, i.e. the first lower priority task to reach a preemption point is preempted. (Note the preempted task may not be the *lowest* priority running task). Alternatively, with a *lazy* approach, preemption is delayed until the *lowest* priority running task reaches a preemption point.

The first analysis for global scheduling with a lazy form of preemption was given by Block *et al.* in 2007 [7] called *link-based* scheduling. Here, when a high priority task is released, and the lowest priority running task τ_i is executing an NPR, then the new task is linked to the processor that is executing τ_i and does not preempt until that task finishes its NPR. (Full details of the method can be found in [7] and section 3.3.3. of Brandenburg's thesis [9]). Brandenburg and Anderson [8] later presented a generic schedulability analysis method that can be applied to different global scheduling algorithms and their schedulability tests.

Building on prior work on optimal fixed priority scheduling with deferred preemption on single processor systems [12], in 2013, Davis *et al.* [14] provided schedulability analysis for global fixed priority scheduling assuming eager preemptions, considering a simple task model with a single final non-preemptive region at the end of each task. They showed that, as in the single processor case, schedulability can be improved by carefully choosing both task priorities and the length of their final non-preemptive regions. Davis *et al.* [16] subsequently corrected a problem with their test. Further, they showed that the eager and lazy approaches to preemption are incomparable *i.e.,* there are task sets that are schedulable using eager preemptions that are unschedulable using lazy preemptions and vice-versa (illustrated by worked examples in the appendix of [16]). Later in 2013, Marinho *et al.* [20] generalized the work of Davis *et al.* [14] to cover task sets with fixed preemption points and hence multiple non-preemptive regions. Marinho *et al.* considered both eager and lazy forms of preemption under global fixed priority scheduling; however, they only gave analysis of the blocking effects, and thus stopped short of providing a full schedulability analysis for eager preemptions. Recently, related work has also been published on global EDF with limited preemption [24], [11].

In this paper, we address the problem of Global Limited Preemption Fixed Priority Scheduling (G-LP-FPS) for tasks with fixed preemption points, and consider how the choice of either an eager or a lazy approach to preemption affects schedulability. The main contributions of the paper are as follows:

1. The introduction of a response time based schedulability test for real-time task sets with fixed preemption points scheduled using G-LP-FPS with eager preemptions. To the best of our knowledge, this is the first such test in the context of G-LP-FPS with fixed preemption points.

2. An evaluation of G-LP-FPS with eager preemptions using the new schedulability test, against G-LP-FPS with lazy preemptions (using link-based scheduling [7]); evaluations shows the that G-LP-FPS with eager preemptions outperforms link-based scheduling for a wide range of settings.

The remainder of the paper contains the system model in Section 2, background in Section 3, the main contributions in Section 4, followed by the evaluations in Section 5 before concluding in Section 6.

2. SYSTEM MODEL

In this section, we present the system model, terminology and definitions assumed in the rest of the paper.

2.1 Task Model

We consider a set of n sporadic real-time tasks $\Gamma = \{\tau_1, \tau_2, ...\tau_n\}$ scheduled on m identical processors. The tasks in Γ are indexed according to their decreasing unique priorities *i.e.,* τ_1 has the highest priority and τ_n the lowest, and each task generates an infinite number of jobs. Let $hp(i)$ denote the subset of tasks with priorities greater than τ_i and $lp(i)$ denote the subset of tasks with priorities lower than τ_i. Each task τ_i is characterized by a minimum inter-arrival time T_i, and a relative deadline $D_i \leq T_i$, and is assumed to contain $q_i \geq 0$ optimal preemption points [21]. The start of the task execution is referred to as the 0^{th} preemption point while the end of the task execution is referred to as the $q_i + 1^{th}$ point; however, preemption of the task is of course not possible at these points. Let $b_{i,j}$, $j = 1...q_i + 1$ denote the worst case execution time of task τ_i between its $j - 1^{th}$ and j^{th} preemption points (The calculation of the WCET between preemption points is an important problem on multicores; but this is not our focus. Instead we focus on the related problem of the prohibitive increase of WCETs due to preemption related overheads). We use the notation $b_{i,j}$ to also refer to the corresponding Non-Preemptive Region (NPR). In any time interval of length t, each task τ_i can be preempted by a higher priority task at most h_i times where $h_i = \sum_{\forall \tau_j \in hp(i)} \left\lceil \frac{t}{T_j} \right\rceil$. Therefore, an upper-bound on the number of preemptions p_i of task τ_i in an interval of length t is given by:

$$p_i = \min(q_i, h_i) \tag{1}$$

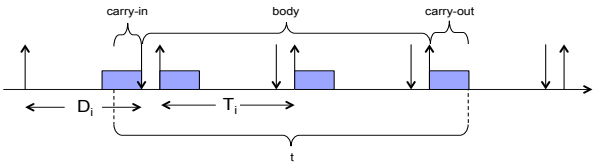

Figure 1: Task with *carry-in* in an interval of length t.

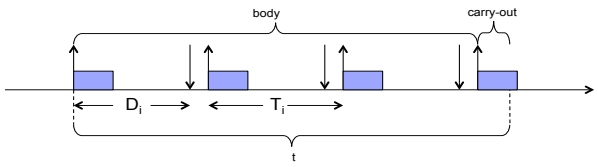

Figure 2: Task without *carry-in* in an interval of length t.

The Worst Case Execution Time (WCET) of each task τ_i can be calculated as $C_i = \sum_{j=1}^{q_i+1} b_{i,j}$. Note that the preemption related overheads can be integrated into the above equation, since we are interested in comparing lazy and eager preemption mechanisms and these overheads are the same in each case, we omit their specific consideration. Similar to Davis *et al.* [16], we also define C_i^* for each τ_i, where $C_i^* = C_i - b_{i,q_{i+1}} + 1$. This is because, when using C_i^* instead of C_i, the resulting response time implies the completion of $C_i - b_{i,q_{i+1}} + 1$ units of execution and hence gives the start time of the final NPR (the +1 ensures start of the final NPR).

2.2 Definitions

According to our model every task is composed of a set of non-preemptive regions with specified lengths. This allows us to define the *preemptability* of a task.

DEFINITION 2.1. *Any task $\tau_i \in \Gamma$ is defined to be **preemptable** at time instant t if and only if time instant t corresponds to a preemption point k, $1 \leq k < q_i$, in the progress of its execution.*

An unfinished task is defined to be *ready* if it is not currently executing (and hence it is assumed to be in the ready queue). Non-preemptive regions within the lower priority tasks can cause priority inversions on higher priority tasks. We enumerate three conditions that are necessary for a *single* priority inversion to occur due to an NPR of a lower priority task under limited preemption scheduling.

DEFINITION 2.2. *A **priority inversion** occurs on an arbitrary task τ_i when the following conditions hold.*

C1: The scheduler is invoked by τ_i, and τ_i is ready.

C2: At least one processor is executing a lower priority task.

C3: All the lower priority jobs are not preemptable.

We differentiate the following types of *interference* for any task τ_i.

DEFINITION 2.3. *The **higher priority interference** on a task τ_i is defined as the cumulative executions of all tasks having a higher priority than τ_i that prevent τ_i from executing on the processor.*

DEFINITION 2.4. *The **lower priority interference** on a task τ_i is defined as the cumulative executions of all tasks having a lower priority than τ_i that prevent τ_i from executing on a processor.*

We use $I_i(t, \tau_j)$ to denote the *interference* on τ_i from a higher priority task τ_j in the time interval of length t and $I_i(t)$ to denote the total *higher priority interference*. Finally, we note that the final NPR of the jobs of any τ_i delays the start of higher priority tasks released during its execution, which may in turn interfere with the next release of τ_i. This interference *pushed through* by any job of a task on to its next job is defined as *push though blocking* (see section 1.4 of [13] for an example).

3. BACKGROUND

In this section, we review the state-of-the-art schedulability analysis for global fixed priority scheduling under the preemptive and limited preemption paradigms.

3.1 Schedulability Analysis for Global Preemptive FPS

Bertogna, Cirinei and Lipari [5] derived an upper-bound on the interference generated by any task τ_j on a lower priority task τ_i over an interval of length t, under Global Preemptive Fixed Priority Scheduling (G-P-FPS) and used this to derive a response time analysis. Even though this analysis has subsequently been improved upon by others, we recall this specific result because of its seminal nature. The upper-bound on the interference generated by any task τ_j on a lower priority task τ_i over an interval of length t is given by:

$$I_i(t, \tau_j) = \min(I_i'(t, \tau_j), t - C_i + 1) \tag{2}$$

where,

$$I_i'(t, \tau_j) = N_j(t)C_j + \min(C_j, t + D_j - C_j - N_j(t)T_j) \tag{3}$$

In the above equation, $N_j(t) = \left\lfloor \frac{t+D_j-C_j}{T_j} \right\rfloor$. Consequently, any task τ_i is schedulable if,

$$R_i = C_i + \left\lfloor \frac{1}{m} \sum_{\tau_j \in hp(i)} I_i(R_i, \tau_j) \right\rfloor \leq D_i \tag{4}$$

Baruah [1] observed that the pessimism in the interference calculation can be attenuated by considering a larger time interval. Baruah also observed that at the time instant earlier than the release of τ_i where at least one processor is idle, the number of jobs that have *carry-in* workload is at most $m - 1$ (see figure 1 and 2 for an illustration of workloads with and without *carry-in*)). The response time analysis of Bertogna, Cirinei and Lipari, was improved by Guan *et al.* [17] by instantiating these observations in the context of G-P-FPS to limit the *carry-in* interference. Later Sun *et al.* [23] identified and fixed some anomalies in the test for the arbitrary deadline case. In order to

determine the schedulability of tasks with fixed preemption points, similar to [16], we define:

$$I_i(t, \tau_j) = \min(I'_i(t, \tau_j), t - C^*_i + 1) \quad (5)$$

where $I'_i(t, \tau_j)$ is given by (3).

3.2 Schedulability analysis with lazy preemption

In this section, we present schedulability analysis for Global Limited Preemption Fixed Priority Scheduling (G-LP-FPS) with lazy preemptions using *link-based* scheduling as proposed by Block *et al.* [7]. First we present an example illustrating lazy preemptions.

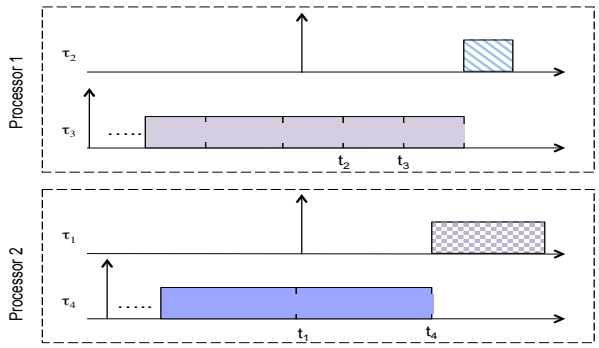

Figure 3: Example schedule illustrating lazy preemption.

EXAMPLE 3.1. *Consider the scenario in Figure 3 where 4 tasks τ_1, τ_2, τ_3 and τ_4 (in decreasing priority order) are executing on two processors. Assume that tasks τ_3 and τ_4 are executing on the processor at time instant t_1. Suppose that τ_1 and τ_2 are released together at time instant t_1, and the scheduling policy uses lazy preemptions. In this case, τ_1 starts executing only at time t_4 even though a lower priority task (τ_3) was available to preempt earlier (at t_2). Moreover, task τ_2 is blocked 3 times by τ_3 since τ_4 has to be preempted first. At time t_4, when the scheduler is invoked, τ_4 is preempted by τ_1. However, since task τ_3 is not preemptible τ_2 still cannot start executing. The total number of such priority inversions could be arbitrarily large if τ_4 has a large non-preemptive region.*

Davis *et al.* [16][14] enumerated a number of interesting observations regarding the blocking introduced by the priority inversions that occur due to non-preemptive execution of lower priority tasks: 1) The number of priority inversions is not limited to the number of processors m, as is the case with Global Non Preemptive Fixed Priority Scheduling (G-NP-FPS) 2) More than one job of the same lower priority tasks can cause priority inversion on a higher priority task 3) More than one non-preemptive region of the same job of each lower priority task can cause priority inversions.

Link based scheduling: Many of these challenges are addressed by *link-based* scheduling developed by Block *et al.* [7] in the context of resource sharing, which is equally applicable to the problem of limited preemption scheduling with NPRs. Link-based scheduling implements a form of lazy preemptions whereby a newly released high priority task is *linked* to the processor executing the lowest priority task. Even though link-based scheduling emulates floating NPR scheduling, it can be used in the context of fixed preemption points scheduling by considering the floating NPRs to be fixed.

Link based scheduling analysis: The analysis for link-based scheduling presented by Block *et al.* [7] and Brandenburg and Anderson [8] provides a simple generic means of accounting for blocking due to NPRs. It uses an *inflation based* method, in which the WCETs of higher priority tasks are inflated to account for blocking from lower priority tasks. Specifically, $\forall \tau_i \in \Gamma$, the associated WCET is inflated as follows:

$$C_i = C_i + \max_{\tau_j \in lp(i)} b_{j,k}, \quad k = 1, \ldots, q_i^1$$

Brandenburg and Anderson [8] proved that the resulting taskset Γ' obtained by inflating the WCETs of all the tasks in Γ with the maximum blocking that they can suffer, is a *safe hard real-time approximation* [9]. This means that if there is a deadline miss with link-based scheduling and NPRs, then there is also guaranteed to be a deadline miss in Γ' under fully preemptive scheduling. Consequently, a fully preemptive schedulability analysis such as that given in [17] can be applied to determine schedulability.

4. SCHEDULABILITY ANALYSIS WITH EAGER PREEMPTIONS

In this section, we present the main contributions of this paper. Specifically, we examine G-LP-FPS with eager preemptions in which the highest priority task is allowed to preempt the *first* lower priority job that becomes preemptable. We show that the number of priority inversions on any task τ_i is upper-bounded by the associated upper-bound on the number of preemptions p_i (defined in (1)). Building on this observation, we derive a response time analysis based test for schedulability under G-LP-FPS with eager preemptions.

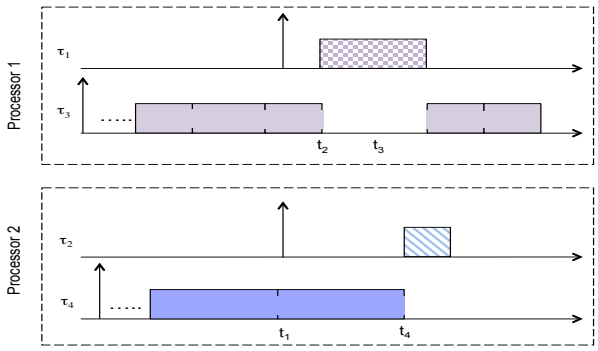

Figure 4: Example schedule illustrating eager preemptions.

In the following we illustrate how G-LP-FPS with eager preemptions schedules the taskset in Example 3.1.

EXAMPLE 4.1. *Consider the same scenario presented in Figure 3 where τ_1 and τ_2 are released at time instant t_1. As with eager preemptions, illustrated in Figure 4, tasks τ_1 and τ_2 are blocked at time instant t_1. When τ_3 finishes executing its non-preemptive region, τ_1 is scheduled on processor* 1.

When τ_4 finishes executing its non-preemptive region, τ_2 can start its execution.

Recall that Example 3.1 showed that with lazy preemption every time a task is inserted into the ready queue after it is released, it may suffer from more than one priority inversion. Eager preemptions enable the *medium* priority task τ_2 to be allocated to the processor earlier than is the case with lazy preemption, as illustrated in Example 4.1. We now derive a bound on the number of priority inversions with eager preemptions.

LEMMA 4.1. *The number of priority inversions on any task τ_i that is inserted into the ready queue under G-LP-FPS with eager preemptions is at most* 1 *before it can again start executing.*

PROOF. Consider the time instant when τ_i is inserted into the ready queue and all the processors are executing the NPRs of lower priority tasks. Let τ_k be the lower priority executing task having the longest duration to the next preemption point from among the executing tasks. When one of the lower priority tasks reaches its preemption point, the scheduler is invoked and the next highest priority task from the ready queue is scheduled. When the next preemption point of τ_k is reached, either τ_i will be (or would have been) scheduled or all processors will be executing tasks having priority higher than τ_i. No *new* non-preemptive regions of lower priority tasks can start executing since there are higher priority tasks waiting in the ready queue. Therefore, the number of priority inversions on τ_i is at most 1 whenever it is inserted into the ready queue. □

We can upper-bound the number of priority inversions on any task using knowledge of the number of preemption points and an upper-bound on the number of preemptions that it can suffer.

COROLLARY 4.1. *The number of priority inversions on any task τ_i under G-LP-FPS with eager preemptions in any time interval of length t is at most p_i, where p_i is defined in (1).*

PROOF. According to Lemma 4.1, τ_i suffers from at most 1 priority inversion every time it is inserted into the ready queue. Recall that each task τ_i can be preempted at most p_i times during any interval of length t. Therefore, τ_i can be inserted into the ready queue most p_i times after it starts, *i.e.,* whenever it is preempted. □

Task execution dynamics on multiprocessors: Phillips *et al.* [22] noted that, under preemptive scheduling, in any time interval of length t between the release time and deadline of a task τ_i, whenever τ_i is not executing, all processors are executing higher priority jobs. On the other hand, when tasks are composed of non-preemptive regions, Phillips *et al.* 's observation needs to be modified as follows: In any time interval of length t, whenever a task τ_i is ready and is not executing, the processor is executing higher priority tasks, or by lower priority NPRs of tasks blocking τ_i. This observation can be applied to obtain the time at which any non-preemptive region of task τ_i can start executing, which in turn gives us the corresponding response time.

In the following two lemmas, we present an upper-bound on the *lower priority interference* on 1) the first NPR of any task τ_i and 2) any other NPR of τ_i.

The *lower priority interference* on the first NPR of τ_i needs to account for the *push through blocking* which is the interference *pushed through* by the final NPR of the previous job of τ_i. However, as noted by Davis *et al.* [16], when calculating the *higher priority interference* using Bertogna *et al.* 's [5] method, since the higher priority tasks are assumed to be executing as late as possible, the effects of *push through blocking* is already accounted for. We therefore, obtain the following upper-bound on the blocking on the first NPR of any τ_i.

LEMMA 4.2. *The* lower priority interference *without accounting for the push through blocking on the first NPR $b_{i,1}$ of a task τ_i over all m processors under G-LP-FPS with eager preemptions is upper-bounded by*

$$\Delta_i^m = \sum \max_{\tau_j \in lp(i)}^{m} \left(\max_{1 \le k \le q_j+1} b_{j,k} \right) \tag{6}$$

where, the $\sum \max_{\tau_j \in lp(i)}^{m}$ term denotes the sum of the m largest values among the NPR's of all $\tau_j \in lp(i)$.

PROOF. This follows from the fact that at most m tasks can be executing at any given time instant, and eager preemptions guarantee that the first preemptable lower priority task is preempted by a higher priority task.

In the worst case, when τ_i is released, all the m processors have just started executing the m largest lower priority NPRs. Consequently, with eager preemptions τ_i needs to wait until these m largest NPRs of lower priority tasks complete their execution before all the processors are busy executing either tasks having higher priority than τ_i (in which case, there is no more priority inversion) or τ_i itself. □

LEMMA 4.3. *The* lower priority interference *on the p^{th} non-preemptive region $b_{i,p}$ of any task τ_i over all m processors under G-LP-FPS with eager preemptions is upper-bounded by*

$$\Delta_i^{m-1} = \sum \max_{\tau_j \in lp(i)}^{m-1} \left(\max_{1 \le k \le q_j+1} b_{j,k} \right) \tag{7}$$

where, $2 \le p \le q_i + 1$ and $\sum \max_{\tau_j \in lp(i)}^{m-1}$ denotes the sum of the $m-1$ largest values among all $\tau_j \in lp(i)$.

PROOF. When τ_i is executing, then at most $(m-1)$ processors are executing lower priority tasks. Suppose that there exists a time instant between the start time and finish time of τ_i when *all* the processors are executing lower priority tasks. Let t denote the *earliest* such time instant. This means that at time instant t, the scheduler scheduled a new low priority job (*i.e.,* an m^{th} lower priority task) even though τ_i was waiting in the ready queue or was executing. We get a contradiction because of our assumption of an global fixed priority based scheduler. □

LEMMA 4.4. *An arbitrary task τ_i can start executing its final NPR q_i+1, in any time interval of length t under G-LP-FPS with eager preemptions if τ_i is ready at the beginning*

of the interval and,

$$C_i^* + \left\lfloor \frac{1}{m} \left(\Delta_i^m + p_i \times \Delta_i^{m-1} + \sum_{\tau_j \in hp(i)} I_i(t, \tau_j) \right) \right\rfloor \leq t \quad (8)$$

where, p_i is given by (1), Δ_i^m is given by (6), Δ_i^{m-1} is given by (7) and $I_i(t, \tau_j)$ is given by (5).

PROOF. Recall that Δ_i^m denotes the sum of the largest lower priority NPRs that can block τ_i over all m processors and $I_i(t, \tau_j)$ gives the worst case interference in the interval t. We know from Lemma 4.1 that the each NPR of τ_i can be blocked at most once. Moreover, we know from lemmas 4.2 and 4.3 that the first NPR of τ_i can be blocked by at most Δ_i^m and that each of the remaining NPRs of τ_i can be blocked at most Δ_i^{m-1} over all the m processors. Moreover, the number of preemptions on τ_i is upper-bounded by p_i. In the worst case, whenever τ_i resumes its execution after a preemption, it is blocked by lower priority NPRs. Therefore the upper-bound on the blocking experienced by τ_i is given by $\Delta_i^m + p_i \times \Delta_i^{m-1}$.

Proof follows from the fact that τ_i has completed $C_i^* = C_i - b_{i,q_i+1} + 1$ units of execution, after incurring the worst case higher and lower priority interference, implying that the final NPR has already started its execution. □

In the following, we present a schedulability test by observing that any task τ_i can be blocked only when it is preempted, and the number of preemptions on τ_i is at most p_i.

THEOREM 4.1. *If for any task τ_i, suppose t' denotes the smallest t for which equation (8) is satisfied, the response time of τ_i under G-LP-FPS with eager preemptions is given by,*

$$R_i = C_i + \left\lfloor \frac{1}{m} \left(\Delta_i^m + p_i \times \Delta_i^{m-1} + \sum_{\tau_j \in hp(i)} I_i(t', \tau_j) \right) \right\rfloor$$

where Δ_i^m and Δ_i^{m-1} are defined in (6) and (7) respectively.

PROOF. The proof follows from the fact that, at t', the final NPR of τ_i has started its execution. □

The smallest t that satisfies (8) can be obtained by first setting $t = C_i^*$ and performing a fixed point iteration on (8) until the condition is satisfied or until a value greater than $D_i - b_{i,q+1} + 1$ is obtained, in which case the task is unschedulable.

The test presented above is, however, pessimistic since it assumes that all the higher priority tasks have *carry-in* jobs that interfere with τ_i. In the following, we build on Baruah's observations [1], and identify a critical scenario that gives the worst case behavior under limited-preemption scheduling while limiting the number of tasks with carry-in jobs. Baruah [1] observed that the worst case scenario that leads to a deadline miss on any job of τ_i, under preemptive scheduling, is such that there exists a busy interval prior to the release of τ_i where all the processors are executing higher priority jobs and extends to the time instant when the job of τ_i can start executing. The start time of such a busy period is assumed to be the earliest time before the release of τ_i such that at least one processor is idle and no processors are idle between the start of the busy period and the release of τ_i. We build on this observation by identifying that, in the case of limited-preemption scheduling, the NPRs of some lower priority jobs could influence the busy period and consider the following scenario.

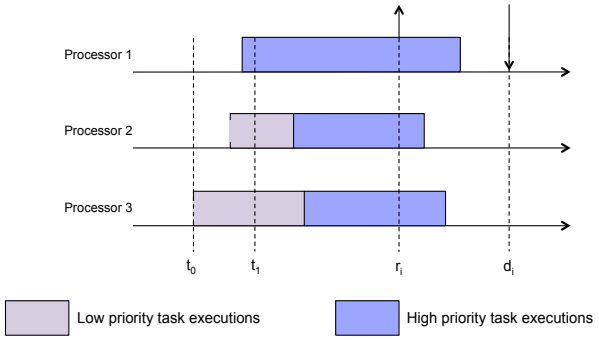

Figure 5: An illustration of the critical scenario.

Critical Scenario: Consider an arbitrary job J_i of a task τ_i released at time instant r_i that has started its execution, such that all jobs of all tasks released prior to r_i are schedulable. We consider the earliest time instant t_1 before the release time r_i at which a higher priority task is released, and is blocked by NPRs of some tasks having a lower priority than τ_i, as illustrated in Figure 5. If there are no higher priority job releases prior to r_i, we set $t_1 = r_i$ (*i.e.,* assume J_i is the task that is blocked). Let the number of processors executing lower priority NPRs at time instant t_1 be x, $1 \leq x \leq m$. Let t_0 denote the earliest among the start times of these x lower priority NPRs that block the higher priority task released at t_1. If there is no such time instant, we set t_0 to be the earliest time instant before r_i at which at least one processor is idle and no processor is idle in (t_0, r_i) (there is always such an instant *e.g.,* at the start of the schedule). Since we consider a sporadic task system, there is always a possibility of lower priority tasks being released, and being executed, at t_0 that may potentially block the execution of τ_i.

OBSERVATION 4.1. *At most $m-1$ higher priority tasks are active at time instant t_0.*

According to our identified critical scenario, at least one processor is starting to execute a lower priority task at time instant t_0. This means that except for the higher priority tasks currently executing on at most $m-1$ processors no other higher tasks are active. If more than $m-1$ higher priority tasks were active, lower priority tasks could not have started executing at t_0.

The above observation allows us to limit the number of *carry-in* tasks to at most $m-1$, rather than assuming that all the tasks have *carry-in* executions. Consequently, we can exploit the recent advances [17] in efficiently accounting for the carry-in interference to determine the schedulability of real-time tasks under G-LP-FPS with eager preemptions. The workload generated by any higher priority task τ_j having *carry-in* interference in any time interval of length t is given as follows [17] (see Figure 1 for an illustration):

$$W_j^{CI}(t) = C_j + \left\lfloor \frac{\max((t - C_j), 0)}{T_j} \right\rfloor \times C_j + \quad (9)$$

$$\min\left([[t - C_j]_0 \ mod \ T_j - (T_j - R_j)]_0, C_j\right)$$

In the above equations, $[A]_B$ means $\max(A, B)$. On the other hand, the workload generated by any higher priority task τ_j that does not *carry-in* interference in any time interval of length t is given as follows [17] (see Figure 2 for an illustration):

$$W_j^{NC}(t) = \left\lfloor \frac{t}{T_j} \right\rfloor \times C_j + \min(t \ mod \ T_j, C_j) \tag{10}$$

Therefore, in any time interval of length t and for any τ_j, we can calculate the upper-bound on the associated *carry-in* as follows [17]: $W_j^{diff}(t) = W_j^{CI}(t) - W_j^{NC}(t)$. A conservative upper-bound on the amount of *carry-in* by higher priority tasks that interferes with any task τ_i can be obtained from the $(m-1)$ largest W_j^{diff}s :

$$I_i^{CI}(t) = \sum \max_{\tau_j \in hp(i)}^{m-1} W_j^{diff}(t)$$

Similarly, a conservative upper-bound on the interference generated by higher priority tasks on τ_i, that do not *carry-in*, in any time interval of length t is given by,

$$I_i^{NC}(t) = \sum_{\tau_j \in hp(i)} W_j^{NC}(t)$$

Therefore, the total higher priority interference on any task τ_i in any time interval of length t can be calculated using:

$$I_i(t) = I_i^{CI}(t) + I_i^{NC}(t) \tag{11}$$

However, when restricting the number of carry-in jobs to at most $m-1$, the tasks are not assumed to be executing as late as possible. Consequently, as noted by Davis *et al.* [16], the effects of *push through blocking* is not automatically accounted for. In the following we show that, for any τ_i, only at most one final NPR of a previous job of τ_i can contribute to the *push through blocking* and the *push through blocking* can affect only its first NPR.

We consider the critical scenario described above and investigate the push through blocking on the job J_i of τ_i.

LEMMA 4.5. *The push through blocking on τ_i comes from at most the final NPR of the previous job of τ_i, under G-LP-FPS with eager preemptions.*

PROOF. According to the critical scenario, all jobs released prior to the release time of the job J_i of τ_i are schedulable, including any previous jobs of τ_i. This means that the final NPR of the previous job of $J\tau_i$ had started executing. Therefore, the push through blocking from its previous NPR must have ended. Therefore, the blocking on J_i comes from only at most the previous job of τ_i and its final NPR. □

LEMMA 4.6. *Only the first NPR of τ_i can be affected by the push through blocking, under G-LP-FPS with eager preemptions.*

PROOF. Follows from the fact that the push through blocking needs to end before the first NPR of τ_i can start executing, and hence does not affect any subsequent NPRs. □

Lemmas 4.5 and 4.6 allows us to upper-bound the lower-priority interference, including any push through blocking, on the first NPR of any τ_i (the blocking on other NPRs of τ_i remains the same as in (7).

LEMMA 4.7. *The* lower priority interference *on the first NPR $b_{i,1}$ of a task τ_i over all the m processors under G-LP-FPS with eager preemptions can be upper-bounded by*

$$\Delta_i^m = \sum \max_{\tau_j \in lp(i)}^{m} (\beta_i) \tag{12}$$

where, set $\beta_i = \{\tau_j \in lp(i), \max_{1 \le k \le q_{j+1}} b_{j,k}\} \cup \{b_{i,q_i+1}\}$, and $\sum \max_{\tau_j \in lp(i)}^{m}$ term denotes the sum of the m largest values.

PROOF. At the release time of τ_i, when calculating blocking, we have two cases:

1. All m processors are executing lower priority NPRs.

   In this case, there is no push through blocking since it must have ended for all the processors to be executing lower priority NPRs. The worst case blocking in this case is given by the m largest lower priority NPRs.

2. At least one processor is executing a higher priority task.

   In this case, the higher priority tasks may bring in push through blocking. Therefore, the worst case blocking on τ_i happens when $m-1$ processors are executing the $m-1$ largest NPRs and the m^{th} processor is executing the highest priority job that brings in a push through blocking. The push through blocking is at most b_{i,q_i+1} according to lemmas 4.5 and 4.6

Therefore, the worst case blocking on the first NPR of any τ_i is obtained by taking the maximum of the two cases described above. □

The Lemma 4.4 can be modified as follows to compute the start time of the final NPR.

LEMMA 4.8. *An arbitrary task τ_i can start executing its final NPR q_i+1, in any time interval of length t under G-LP-FPS with eager preemptions if τ_i is ready at the beginning of the interval and,*

$$C_i^* + \left\lfloor \frac{1}{m} \left(\Delta_i^m + p_i \times \Delta_i^{m-1} + I_i(t) \right) \right\rfloor \le t \tag{13}$$

where, p_i is given by (1), Δ_i^m is given by (12), Δ_i^{m-1} is given by (7) and $I_i(t)$ is given by (11).

Consequently, the response time of any $\tau_i \in \Gamma$ can be calculated by modifying Theorem 4.1, as follows:

THEOREM 4.2. *For any task τ_i, suppose t' denotes the smallest t for which equation (13) is satisfied, the response time of τ_i under G-LP-FPS with eager preemptions is given by,*

$$R_i = C_i + \left\lfloor \frac{1}{m} \left(\Delta_i^m + p_i \times \Delta_i^{m-1} + \sum_{\tau_j \in hp(i)} I_i(t') \right) \right\rfloor$$

where Δ_i^m and Δ_i^{m-1} are defined in (12) and (7) respectively and $I_i(t')$ is defined in (11).

While it may seem that accounting for worst case *lower priority interference* per preemption on each task is pessimistic, it is sufficient to guarantee the absence of scheduling anomalies. For example, it may happen that some higher priority tasks execute for less than their worst case execution time and a lower priority task starts executing a large NPR in the resulting slack.

5. EVALUATIONS

In this section, we report the results of an experimental evaluation of the performance of G-LP-FPS with eager and lazy preemptions using *weighted schedulability* [2]. For this purpose, we used the test derived in this paper for the eager preemption approach (EPA). The schedulability under lazy preemption approach (LPA) was determined for link-based scheduling [7] using the inflation based approach [8] in conjunction with the schedulability test for G-P-FPS given by Guan *et al.* [17]. We calculated the weighted schedulability variations with respect to 1) number of tasks per taskset 2) the NPR lengths and 3) number of processors. Specifically, we varied:

1. the number of tasks keeping the number of processors and NPR lengths constant
2. the NPR lengths keeping the number of processors and the number of tasks constant for tasks with 1) long NPRs relative to their WCET and 2) short NPRs relative to their WCET
3. the number of processors keeping the number of tasks and NPR lengths constant

A higher weighted schedulability implies a better scheduling algorithm since the schedulability is weighted against taskset utilizations. For *reference*, we also included the weighted schedulabilities under G-P-FPS assuming no overheads and G-NP-FPS; the performance of G-P-FPS will significantly decrease relative to the other algorithms when overheads are included since the pessimism associated with overhead accounting is much higher. Detailed evaluation via analysis including overheads and measurements from a real implementation are future work. Moreover, the behavior of G-P-FPS can be obtained using G-LP-FPS by allowing preemptions after every unit of execution, with ties broken using task priority, and that of G-NP-FPS can be obtained by having no preemption points.

5.1 Experimental Setup

We used the *UUnifast-Discard* algorithm proposed by Davis and Burns [15] to generate task utilizations. The minimum separation times (periods) were uniformly generated between 50 and 500 (note that the time period ranges are changeable). Deadlines were set equal to periods (implicit deadlines), although the schedulability tests also apply to constrained-deadline tasksets. The largest NPR values for each $\tau_i \in \Gamma$ were set as a percentage of its WCET denoted by $\mathcal{P}$ (ceiling function was applied to get integer values). We assumed that all NPRs of τ_i have the same length equal to the largest value, except for the first that can be smaller (depending on the WCET). Note that prior work [14] shows that larger final NPRs give improved schedulability. The utilization of the tasksets ranged from a minimum of U_{tot}^{min} to a maximum of $U_{tot}^{max} = m$, where m is the number of processors. In the experiments, we set $U_{tot}^{min} = 2.4$ since we are more interested in scheduling tasks with larger utilizations to effectively use the platform. We assumed Deadline Monotonic Priority ordering (DMPO) [19]; although better priority assignment algorithms exists for G-P-FPS DMPO serves our purpose of comparison since we use the same priority assignment for all the considered scheduling algorithms. The test in [17] was adopted as reference for G-P-FPS, and for G-NP-FPS, we used our test after setting the largest NPR length equal to the task computation times.

5.2 Experimental Results

In the first experiment, we examined the performance of G-LP-FPS for varying numbers of tasks for $m = 4$ processors and $\mathcal{P} = 5\%$. The results of the experiments are illustrated in Figure 6.

The experiment indicates that for high utilization tasksets with large numbers of tasks, G-LP-FPS with eager preemptions performs better than with lazy preemptions. This is due to the inherent pessimism in the inflation based technique, which is amplified for large tasksets and large utilizations. We also observe that the performance of G-NP-FPS improved significantly with an increasing number of tasks. The main reason is that when the number of tasks is large, the individual task utilizations are very small, hence most tasks have relatively small computation times in relation to their deadlines and so are more amenable to G-NP-FPS [18]. By contrast, the presence of preemption points introduces additional blocking (from *lower priority interference* occurring after the start of the execution) leading to reduced schedulability. Evaluations for larger numbers of processors (specifically m=6 and $m = 8$), varying the number of tasks showed similar behavior[1].

We also investigated the consequences of changing NPR lengths, and consequently the number of preemption points, on schedulability. Increasing NPR lengths increases pessimism in link-based scheduling because of the inflation of WCETs. However, it can have beneficial effects in our test for G-LP-FPS under eager preemption. We varied the size of NPRs as a percentage of the corresponding WCETs, between 5% and 100% (approximated using a ceiling function). The results are presented in Figure 7.

As can be seen the schedulability varies in a saw-tooth manner until the NPR lengths reach 50% (*i.e.,* the number of preemption points reduces to 1). Once past 50%, the schedulability continues to decrease until the tasks are almost fully non-preemptive, after which the schedulability starts to increase again. This is because, once the NPR lengths go past 50% of WCET, there is no further reduction in the number of preemption points; however, the length of the largest NPRs continues to increase, consequently increasing blocking on higher priority tasks (that may have a single preemption point). Finally, when the tasks become fully non-preemptive, schedulability increases since the *lower priority interference* decreases due to the reduction in the number of preemption points from one to zero.

The schedulability varies in a saw-tooth fashion between 5% and 50% demonstrating the futility of increasing the largest NPR lengths without reducing the number of preemption points. For example, as we increase the largest NPR lengths from 25% to 30%, there is no further decrease in the number of preemption points (which is 3 for both cases). On the other hand, the NPR lengths increase leading to increased *lower priority interference* that occurs after the start of the task executions, reducing

[1]see additional graphs available online at `http://www.idt.mdh.se/~atl05/pdf_files/graphs-G-LP-FPS-2015.pdf`

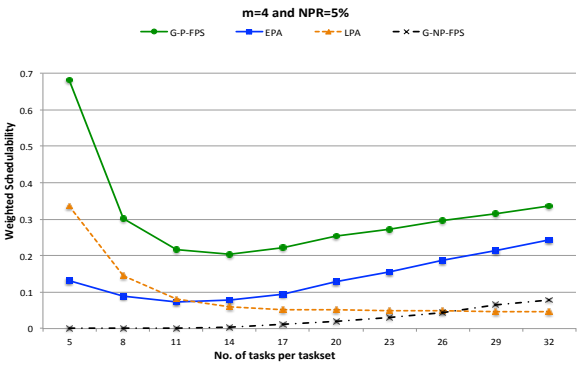

Figure 6: Varying number of tasks.

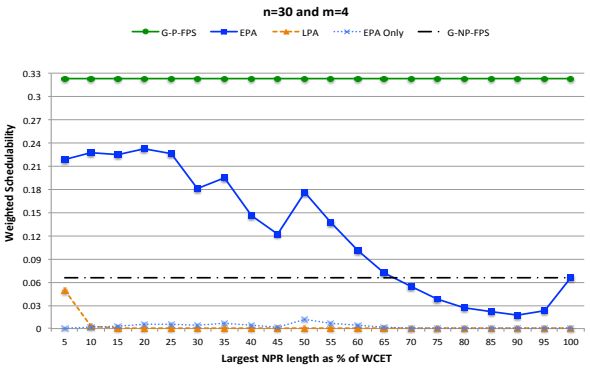

Figure 7: Varying NPR lengths (large NPRs).

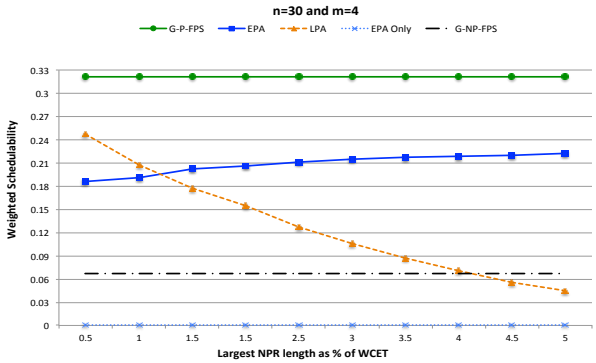

Figure 8: Varying NPR lengths (small NPRs).

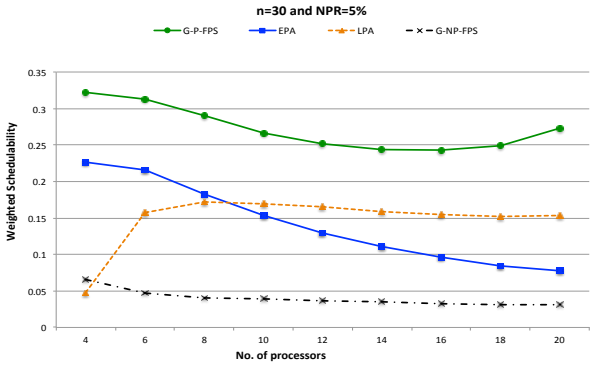

Figure 9: Varying number of processors.

schedulability. On further increasing the largest NPR lengths to 35%, the number of preemption points decreases to 2, consequently improving schedulability due to reduction of *lower priority interference* after the start time of the tasks. The same reasoning explains the decrease in schedulability when increasing the largest NPR lengths from 35% to 45% since the number of preemptions remain unchanged at 2. When the NPR lengths increase to 50%, schedulability increases because the number of preemptions decrease to 1. In a few cases, our experiment identified some tasksets as schedulable using G-LP-FPS with eager preemptions but unschedulable under any other approach. However, this was not observed with link-based scheduling since it is based on the test for G-P-FPS [17] after inflating the WCETs. Link based scheduling fared better compared to G-LP-FPS with eager preemptions for very small NPR lengths as shown in Figure 8.

Finally, we varied the number of processors keeping the number of tasks and NPR lengths constant, at 30 and 5% respectively (reported in Figure 9). We observe that link-based scheduling has a higher schedulability when the number of processors increases. This is due to the fact that availability of more processors implies better schedulability under link-based scheduling for low utilization tasksets with a fixed number of tasks. The same trend is seen in the left end of Figure 6 where the number of tasks compared to the available number of processors is small. However, for tasksets with a large number of tasks and high utilizations, the eager preemption approach is the most effective (Figure 6).

Even though the focus of the experiments presented in this paper is on schedulability, we note that, preliminary experiments simulating G-LP-FPS with eager and lazy preemptions suggest that the number of preemptions observed during run-time is slightly higher in the case of eager preemptions. This happens because with lazy preemptions, further higher priority tasks may be released before preemption can occur, thus the tasks are more likely to run in priority order with fewer preemptions.

6. CONCLUSIONS

Limiting preemptions to predetermined points within real-time tasks is an effective means of reducing preemption and migration related overheads. However, it introduces an interesting question of how best to manage preemption. At one extreme, the scheduler can choose *eager* preemption of the first executing lower priority task that becomes preemptable, while at the other extreme, it can restrict preemption to only the lowest priority executing task, when it becomes preemptable, referred to as *lazy* preemption. Each strategy has a different effect in terms of the number of priority inversions in the schedule, that in turn affects schedulability.

In this paper, we made the following contributions:

1. We derived a schedulability analysis for G-LP-FPS with eager preemptions building on the observation that blocking happens only when a task resumes

execution. To the best our knowledge, this is the first such test for G-LP-FPS with fixed preemption points.

2. We evaluated the new schedulability test by comparing it with a test for G-LP-FPS with lazy preemption. The test used was the state-of-the-art test for G-P-FPS [17] supplemented by the inflation-based approach of accounting for blocking. [7]. Our evaluations showed that G-LP-FPS with eager preemptions outperforms link-based scheduling in the context of fixed preemption points.

It remains to be seen whether the eager preemption approach is beneficial in the context of floating NPRs. Future work includes investigating optimal preemption point placement strategies, as well as experiments considering overheads on real platforms.

Acknowledgments

We would like to thank Björn Brandenburg for his insightful comments on the paper. This work was funded in part by the EPSRC project MCC (EP/K011626/1), and by the European Union, under the Seventh Framework Programme (FP7/2007-2013), grant agreement no 611016 (P-SOCRATES). EPSRC Research Data Management: No new primary data was created during this study.

7. REFERENCES

[1] S. Baruah. Techniques for multiprocessor global schedulability analysis. In *The 28th IEEE International Real-Time Systems Symposium*, 2007.

[2] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *The International Workshop OSPERT*, 2010.

[3] M. Bertogna. Real-time scheduling analysis for multiprocessor platforms. In *PhD Thesis, SSSUP, Pisa*, May 2008.

[4] M. Bertogna, G. Buttazzo, M. Marinoni, G. Yao, F. Esposito, and M. Caccamo. Preemption points placement for sporadic task sets. In *The Euromicro Conference on Real-Time Systems*, 2010.

[5] M. Bertogna, M. Cirinei, and G. Lipari. New schedulability tests for real-time task sets scheduled by deadline monotonic on multiprocessors. In *Principles of Distributed Systems*, Lecture Notes in Computer Science. 2006.

[6] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazzo. Optimal selection of preemption points to minimize preemption overhead. In *The 23rd Euromicro Conference on Real-Time Systems*, 2011.

[7] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *The 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2007.

[8] B. Brandenburg and J. Anderson. A clarification of link-based global scheduling. In *Technical Report MPI-SWS-2014-007*, 2014.

[9] B. B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, UNC Chapel Hill, 2011.

[10] G. Buttazzo, M. Bertogna, and G. Yao. Limited preemptive scheduling for real-time systems: A survey. *The IEEE Transactions on Industrial Informatics*, 2012.

[11] B. Chattopadhyay and S. Baruah. Limited-preemption scheduling on multiprocessors. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*. ACM, 2014.

[12] R. Davis and M. Bertogna. Optimal fixed priority scheduling with deferred pre-emption. In *The Real-Time Systems Symposium*, 2012.

[13] R. Davis, A. Burns, R. Bril, and J. Lukkien. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 2007.

[14] R. Davis, A. Burns, J. Marinho, V. Nelis, S. Petters, and M. Bertogna. Global fixed priority scheduling with deferred pre-emption. In *The International Conference on Embedded and Real-Time Computing Systems and Applications*, 2013.

[15] R. I. Davis and A. Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Systems*, 2011.

[16] R. I. Davis, A. Burns, J. Marinho, V. Nelis, S. M. Petters, and M. Bertogna. Global and partitioned multiprocessor fixed priority scheduling with deferred preemption. *ACM Transactions on Embedded Computing Systems*, 2015.

[17] N. Guan, M. Stigge, W. Yi, and G. Yu. New response time bounds for fixed priority multiprocessor scheduling. In *The 30th IEEE Real-Time Systems Symposium*, 2009.

[18] N. Guan, W. Yi, Z. Gu, and G. Yu. New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms. In *The IEEE International Real-time Systems Symposium*, 2008.

[19] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 1982.

[20] J. Marinho, V. Nelis, S. Petters, M. Bertogna, and R. Davis. Limited pre-emptive global fixed task priority. In *The International Real-time Systems Symposium*, 2013.

[21] B. Peng, N. Fisher, and M. Bertogna. Explicit preemption placement for real-timeconditional code. In *The Euromicro Conference on Real-Time Systems*, July 2014.

[22] C. A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation (extended abstract). In *The ACM symposium on Theory of computing*, 1997.

[23] Y. Sun, G. Lipari, N. Guan, and W. Yi. Improving the response time analysis of global fixed-priority multiprocessor scheduling. In *The 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2014.

[24] A. Thekkilakattil, S. Baruah, R. Dobrin, and S. Punnekkat. The global limited preemptive earliest deadline first feasibility of sporadic real-time tasks. In *The 26th Euromicro Conference on Real-Time Systems*, July 2014.