Asymptotically Optimal Multiprocessor Real-Time Locking for non-JLFP Scheduling

Zelin Tong, Syed Ali, and James H. Anderson
Department of Computer Science
University of North Carolina at Chapel Hill

***Abstract*—In prior work, a number of asymptotically optimal suspension-based real-time locking protocols have been presented for job-level fixed priority (JLFP) schedulers, where job priorities do not change. However, the optimality proofs for these locking protocols break down under non-JLFP scheduling, where job priorities can vary. In fact, the problem of designing an asymptotically optimal real-time locking protocol for general non-JLFP scheduling has remained open. This paper closes this problem by presenting the *non-JLFP locking protocol (NJLP)*, the first asymptotically optimal suspension-based real-time locking protocol for non-JLFP schedulers.**

I. Introduction

Real-time scheduling algorithms can generally be classified into two categories: those that maintain fixed job priorities and those that allow job priorities to change. The former category, known as *job-level fixed priority (JLFP) schedulers*, includes algorithms like the global earliest-deadline-first (G-EDF) and global fixed-priority (G-FP) schedulers. The latter category, known as *non-JLFP schedulers*, includes algorithms such as the PD^2 P-fair scheduler [3], the earliest-deadline-zero-laxity (EDZL) scheduler [12], and the earliest-eligible-virtual-deadline-first (EEVDF) scheduler [19], which has recently been incorporated into the Linux kernel.

In addition to non-JLFP schedulers, various scheduling techniques used in conjunction with JLFP schedulers also allow job priorities to change, thus emulating the behavior of non-JLFP schedulers. These techniques include job splitting [15], where a job is divided into multiple sub-jobs with different priorities, and server-based scheduling [20], where jobs can change priorities when migrating between servers.

Benefits of non-JLFP schedulers. Non-JLFP schedulers and scheduling techniques, which we henceforth collectively refer to as *non-JLFP scheduling*, can improve the schedulability of task systems by relaxing job priority restrictions. For instance, the non-JFLP PD^2 P-fair scheduler is optimal for scheduling implicit-deadline sporadic tasks on multiprocessors [3]. Additionally, global EDZL has been shown to strictly dominate the comparable JLFP scheduler G-EDF in terms of schedulability [12], meaning that there exist task sets schedulable by global EDZL and not by G-EDF, but not vice versa. Additionally, utilizing non-JLFP scheduling techniques such as job splitting can enable lower job response times [15].

Work was supported by NSF grants CPS 2333120, CPS 2038855, CPS 2038960, and CNS 2151829

Supporting shared resources. Real systems typically have shared resources such as shared data structures or hardware devices that require mutually exclusive (mutex) access. Mutex sharing can be realized by using suspension-based real-time locking protocols. Under such a protocol, a job that requests an unavailable resource is suspended until its request can be satisfied. If a suspended job has sufficient priority to be scheduled, it incurs *priority-inversion blocking (pi-blocking)*. Pi-blocking impinges on schedulability by delaying job completions. As such, the worst-case pi-blocking that can be experienced by a job when it requests a resource is commonly used as a metric in comparing real-time locking protocols. In this paper, we consider this metric in the context of *suspension-oblivious (s-oblivious)* schedulability analysis, where pi-blocking time is analytically treated as execution time.

Optimal pi-blocking results under JLFP scheduling. Under JLFP scheduling, optimality results for real-time locking protocols are well known. Assuming s-oblivious analysis, any m-processor mutex locking protocol under a JLFP scheduler is subject to a pi-blocking lower bound of $\Omega(m)$ request lengths [2], [7]. Moreover, numerous locking protocols have been proposed for which maximum pi-blocking is $O(m)$ request lengths [7]–[9], making them *asymptotically optimal*.

Locking under non-JLFP scheduling. In contrast, no optimality results have been established for real-time locking under non-JLFP scheduling. Indeed, the optimality proofs for prior protocols [7]–[9] depend crucially on having fixed job priorities. Thus, these optimal locking protocols for JLFP scheduling are not necessarily optimal for non-JLFP scheduling. While some prior work on locking under *specific* non-JLFP schedulers exists [15], [16], those works do not yield pi-blocking optimality results. Thus, the problem of designing an asymptotically optimal locking protocol for non-JLFP schedulers remains open.

Contributions. In this paper, we close this problem. We do so by first establishing a pi-blocking lower bound for *any* locking protocol under non-JLFP scheduling of $\Omega(m+m(H_n-H_m))$ request lengths, where H_i denotes the i^{th} harmonic number. This result shows that *it is pointless to try to design a generally applicable mutex locking protocol to be used in the considered context with asymptotically better pi-blocking because no such protocol exists*. Next, we introduce the *non-JLFP locking protocol (NJLP)*, and prove that it is asymptotically optimal

under non-JLFP scheduling. The existence of this protocol has the side effect of showing that our lower bound is *asymptotically tight*. The NJLP is not merely of theoretical interest. To demonstrate this, we present the results of an overhead-award schedulability study, where said overheads under the NJLP were measured in an actual implementation of it. In this study, usage of the NJLP resulted in comparable and often better schedulability than a protocol that is simpler but not optimal.

Organization. In the following sections, we establish these results by first providing necessary background information (Sec. II), and by then presenting our pi-blocking lower-bound proof (Sec. III), the NJLP and its pi-blocking analysis (Sec. IV), and our experimental evaluation of it (Sec. V).

II. System Model and Background

We consider a task system composed of n sporadic tasks $\tau_1, \tau_2, ..., \tau_n$ executing on m identical processors where $n > m$.[1] Each task τ_i releases a series of jobs $J_{i,1}$, $J_{i,2}$, ... and is characterized by its *period* T_i, *worst case execution time* (WCET) C_i, and *relative deadline* D_i. In this paper, we consider constrained deadline tasks, where each $D_i \leq T_i$. We denote the release time of a job $J_{i,j}$ as $r_{i,j}$ and its completion time as $f_{i,j}$. We assume time to be continuous. After a job $J_{i,j}$ is released at time $r_{i,j}$, we say that $J_{i,j}$ is *pending* until it completes at time $f_{i,j}$. A pending job is either *ready* to be scheduled, or *suspended*, where it cannot be scheduled.

Scheduling. We consider jobs scheduled under a clustered scheduler, where each job is assigned to one of m/c clusters, each containing c processors.[2] For notational brevity, we assume that c divides m. In clustered scheduling, a job is scheduled when its priority is among the c highest in its cluster. Jobs are free to migrate among processors in the same cluster, but may not migrate across clusters. Since partitioned scheduling (resp. global scheduling) is a special case of clustered scheduling with $c = 1$ (resp. $c = m$), our work is applicable to both partitioned and global schedulers. We assume non-JLFP scheduling, as defined next.

Definition 1. *Under non-JLFP scheduling, the priority of a job $J_{i,j}$ can change at any time t in the time interval $[r_{i,j}, f_{i,j})$.*

Example schedulers that satisfy Def. 1 are PD^2 P-fair [3] and EEVDF [19] (replaced the Completely Fair Scheduler as the main scheduler in Linux in version 6.6 [13]). Additionally, certain scheduling techniques used alongside JLFP schedulers can also be classified as non-JLFP scheduling under Def. 1. Examples of such techniques include job splitting [15], which divides a job into sub-jobs with different priorities, and select server-based scheduling techniques [20], which allow jobs to change priorities when migrating between servers.

Resource model. We consider a set of q resources $\ell_1, \ell_2, ..., \ell_q$ where each resource ℓ allows for mutex access, i.e., ℓ can be held by at most one job at a time. Each job $J_{i,j}$ contains $N_{i,\ell}$ requests for resource ℓ. For each request $\mathcal{R}$, we let $J(\mathcal{R})$ denote the job that contains $\mathcal{R}$ and $L(\mathcal{R})$ denote the length of $\mathcal{R}$'s resource access. When $\mathcal{R}$ is *issued*, $J(\mathcal{R})$ suspends until it is allowed exclusive access to ℓ, upon which $\mathcal{R}$ becomes *satisfied*. When $J(\mathcal{R})$ relinquishes exclusive access to ℓ, we say that $\mathcal{R}$ *completes*. $\mathcal{R}$ is *active* from the time it was issued until the time it completes. We let L_ℓ denote the maximum length of any request's resource access for ℓ. Additionally, we do not allow a job to request more than one resource at a time.

[1] We assume $n > m$ to simplify notation. However, our analysis in Sec. III and Sec. IV can be trivially extended to include the case where $n \leq m$.

[2] While our work can apply to schedulers with differing cluster sizes, we assume equal cluster sizes to simplify notation.

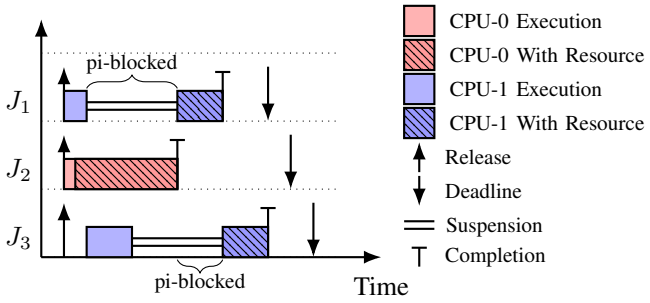


Fig. 1: A schedule of the three jobs in Ex. 1 that illustrates when jobs are pi-blocked.

Priority-inversion blocking. Priority-inversion blocking (pi-blocking) occurs when a job cannot be scheduled despite having sufficient priority. This occurs when a job issues a request for a resource that cannot be immediately satisfied and suspends. Formally, the definition of pi-blocking is dependent upon the type of schedulability analysis used. For suspension-based locks, such analysis can either be *suspension-oblivious* or *suspension-aware* [7]. In this paper, we focus on the former, where job suspensions are analytically treated as job execution. Thus, we use the pi-blocking definition in [7].

Definition 2. *Under s-oblivious schedulability analysis, a job $J_{i,j}$ incurs pi-blocking at time t if $J_{i,j}$ is pending but not scheduled and fewer than c higher-priority jobs are pending in $J_{i,j}$'s cluster.*

Under Def. 2, an unscheduled job is pi-blocked if it is among the c highest-priority pending jobs in its cluster. This is demonstrated by the following example depicted in Fig. 1.

Example 1. *Consider three jobs, J_1, J_2, J_3, sharing one resource scheduled on a cluster of two processors ($c = 2$). In the schedule of the three jobs depicted in Fig. 1, J_i has the i^{th} highest priority in the cluster. When J_1 suspends, it cannot be scheduled despite being the highest-priority job in the cluster. Therefore, J_1 incurs pi-blocking while it is suspended. Meanwhile, when J_3 suspends, it is only pi-blocked if there are fewer than $c = 2$ higher-priority pending jobs. Therefore, J_3 is only pi-blocked when J_2 completes, leaving J_3 with only one higher-priority pending job in the cluster.*

Under s-oblivious schedulability analysis, the pi-blocking experienced by a job due to a locking protocol can be accounted for by inflating job execution times.

Progress mechanisms. A real-time locking protocol must limit the pi-blocking experienced by a job. To this end, if a resource ℓ is held by some job $J_{i,j}$ that is causing other jobs to be pi-blocked, $J_{i,j}$ must *make progress* on the completion of its request. However, since $J_{i,j}$ may not have sufficient priority to be scheduled, a *progress mechanism* must be used alongside the locking protocol to ensure that $J_{i,j}$ is scheduled. Such a mechanism must ensure the following property.

Property 1. *Let $W(\mathcal{R})$ denote the set of requests waiting on the completion of request $\mathcal{R}$. If $\mathcal{R}$ is satisfied at time t, and there exists a request $\mathcal{R}' \in W(\mathcal{R})$ where $J(\mathcal{R}')$ is pi-blocked, then $J(\mathcal{R})$ is scheduled.*

Many progress mechanisms have been proposed for locking protocols under JLFP scheduling [8], [10], [18]. In this paper, we consider two progress mechanisms that are easily adaptable for use with locking protocols under non-JLFP scheduling. The first is *migratory priority inheritance*, where the resource-holding job is scheduled "in place" of a suspended job that has sufficient priority to be scheduled.

Definition 3. *Under migratory priority inheritance, whenever request $\mathcal{R}$ is satisfied but $J(\mathcal{R})$ is ready but not scheduled, and there exists a job $J(\mathcal{R}')$ where $\mathcal{R}' \in W(\mathcal{R})$ such that $J(\mathcal{R}')$ has sufficient priority to be scheduled, $J(\mathcal{R})$ migrates to $J(\mathcal{R}')$'s cluster and assumes $J(\mathcal{R}')$'s priority. After $\mathcal{R}$ completes, $J(\mathcal{R})$ migrates back to its original cluster [10].*

Migratory priority inheritance allows for a suspended job to ensure the progress of a resource-holding job from a different cluster, which makes it a suitable progress mechanism for a locking protocol under clustered scheduling. However, having to migrate jobs from cluster to cluster may cause migratory priority inheritance to incur large overheads [10].

In contrast, *priority inheritance* allows a resource-holding job to make progress by inheriting the priority of a suspended job. This mechanism can guarantee the progress of a resource-holding job under global scheduling (but not necessarily clustered scheduling), and it has lower overheads than migratory priority inheritance. Under global scheduling, priority inheritance works as follows.

Definition 4. *Under priority inheritance, if request $\mathcal{R}$ is satisfied, $J(\mathcal{R})$ is ready but not scheduled, and there exists a job $J(\mathcal{R}')$ where $\mathcal{R}' \in W(\mathcal{R})$ such that $J(\mathcal{R}')$ has sufficient priority to be scheduled, then $J(\mathcal{R})$ inherits the priority of the highest-priority job that has issued a request in $W(\mathcal{R})$.*

Demonstrating asymptotically optimal pi-blocking. To demonstrate that the maximum pi-blocking due to a locking protocol is asymptotically optimal under non-JLFP scheduling, we must first determine a lower bound on the maximum pi-blocking due to a single resource request that is applicable to any locking protocol under non-JLFP scheduling. We refer to this lower bound as a *pi-blocking lower bound* under non-JLFP scheduling. Since no locking protocol can achieve a lower maximum per-request pi-blocking bound than this lower bound, a locking protocol that ensures a maximum per-request pi-blocking bound that is asymptotically equivalent to the lower bound is *asymptotically optimal*.

III. Lower Bound

In this section, we derive a lower bound on the maximum pi-blocking of any suspension-based mutex locking protocol; this amount of pi-blocking occurs in a schedule as allowed by global non-JLFP scheduling.[3] To quantify our pi-blocking lower bound, we first require the notion of *harmonic numbers*.

Definition 5. *The i^{th} harmonic number, denoted as H_i, is given by $H_i = \sum_{x=1}^{i} \frac{1}{x}$*

We establish a pi-blocking lower bound of $\Omega(m + m(H_n - H_m))$ request lengths. We do this by demonstrating the existence of a pathological task set and series of priority changes such that some job must incur $\Omega(m + m(H_n - H_m))$ pi-blocking per resource request under an arbitrary suspension-based mutex locking protocol in a particular schedule allowed by global non-JLFP scheduling. We detail our pathological task set and priority changes in Sec. III-A and provide our lower bound proof in Sec. III-B.

A. Pathological Task Set and Schedule

We let $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$ be our pathological task set of n tasks to be globally scheduled on m processors. Each $\tau_i \in \Gamma$ has an implicit deadline ($D_i = T_i$), a WCET of 1 ($C_i = 1$), and some sufficiently large period $T_i = T$ such that Γ is schedulable under any non-JLFP scheduling method.

Release and request times. For our lower-bound proof in Sec. III-B, we need only to analyze the first job of each task. This is due to our construction of Γ, which causes the behavior of subsequent jobs of a task to mirror that of the first. Thus, we focus only on the release and request issuance times of each task's first job, given by the following rules.

L1. For each $\tau_i \in \Gamma$, $J_{i,1}$ is released at $t = 0$.
L2. Upon release, $J_{i,1}$ immediately issues a request $\mathcal{R}_i$ for resource ℓ with a request length $L(\mathcal{R}_i) = 1$.

To aid in the pi-blocking lower-bound analysis in Sec. III-B, we order the requests of each $J_{i,1}$ by the time they become satisfied in the considered arbitrary locking protocol.

Definition 6. *Let $\mathcal{R}^i$ denote the i^{th} request to become satisfied among $\{\mathcal{R}_x \mid \tau_x \in \Gamma\}$ by the considered locking protocol. We denote the time $\mathcal{R}^i$ becomes satisfied as t_i.*

Since at most one request can be satisfied at any time under a mutex locking protocol, Rule L2 and Def. 6 ensure the following property.

Property 2. *$t_{i+1} \geq t_i + 1$ for $i \in [1, n)$.*

Additionally, due to Def. 6, we can determine the status of a request $\mathcal{R}^j$ at t_i. For example, if $j < i$, then since resource

[3]We analyze global scheduling to avoid cumbersome notation. The proof that our asymptotic bound holds under clustered scheduling is given in the appendix [21].

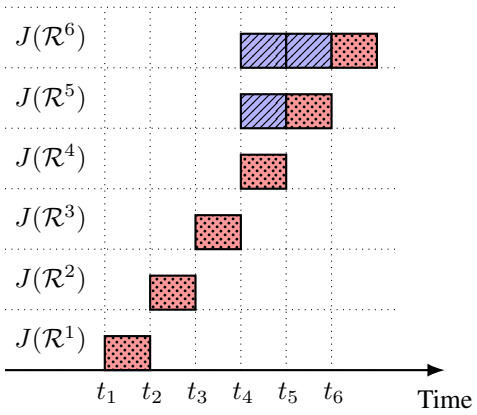


Fig. 2: Example schedule of Γ in the time interval $[t_1, t_n)$. Red indicates times when the job of the satisfied request can be scheduled and blue indicates times when a job is pi-blocked by Rule L3.

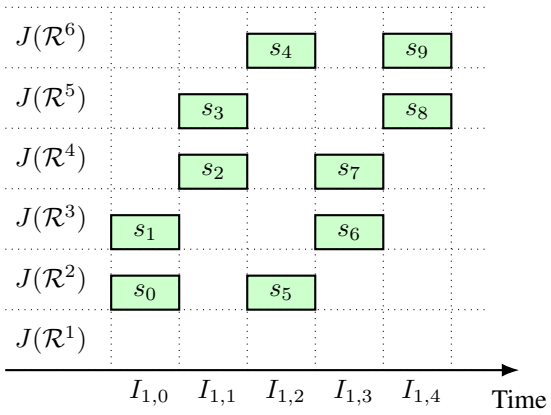


Fig. 3: Example schedule of Γ in the time interval $[t_1, t_2)$ subdivided into $n-i$ sub-intervals $I_{1,0}$ to $I_{1,4}$. Green indicates the time when a job is pi-blocked by Rule L4 and is labeled by the slot the job fills.

ℓ only allows mutex access, $\mathcal{R}^j$ must be complete when $\mathcal{R}^i$ becomes satisfied at t_i. Conversely, if $j > i$, then $\mathcal{R}^j$ will become satisfied at t_j, thus it is unsatisfied at t_i. This leads to the following lemma.

Lemma 1. *If $j > i$, then $J(\mathcal{R}^j)$ is suspended in the time interval $[t_i, t_{i+1})$.*

Proof. By Def. 6, $\mathcal{R}^j$ is unsatisfied until t_j. Since $j > i$, $\mathcal{R}^j$ is unsatisfied in the time interval $[t_i, t_{i+1})$. Therefore, $J(\mathcal{R}^j)$ is suspended in $[t_i, t_{i+1})$. □

Pathological priority changes. To ensure that $\Omega(m+m(H_n - H_m))$ pi-blocking occurs under non-JLFP scheduling, a series of orchestrated job priority changes must occur. We describe these priority changes using Rules L3 and L4, given below.

> **L3.** In each time interval $[t_i, t_{i+1})$ where $i \in [n-m, n)$, $J(\mathcal{R}^j)$'s priority where $j \in (i, n]$ becomes the $(j-i)^{th}$ highest among all pending jobs.

We illustrate the effects of Rules L1 to L3 on the schedule of our task set Γ in the following example, depicted in Fig. 2.

Example 2. *Consider the pathological task set $\Gamma = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6\}$ globally scheduled on two processors. In this scenario, $n = 6$ and $c = m = 2$. By Rules L1 and L2, $\mathcal{R}^1$ to $\mathcal{R}^6$ are issued simultaneously at $t = 0$. Due to Def. 6, request $\mathcal{R}^i$ becomes satisfied at t_i, and must be complete by t_{i+1} when $\mathcal{R}^{i+1}$ becomes satisfied. This pattern is illustrated in Fig. 2, where jobs with satisfied requests are colored red. Rule L3 begins taking effect at $t_{n-m} = t_4$. In $[t_4, t_5)$, $J(\mathcal{R}^5)$ obtains the highest priority, and $J(\mathcal{R}^6)$ obtains the 2^{nd} highest priority. Since both jobs are suspended due to Lem. 1, both jobs are pi-blocked in $[t_4, t_5)$ according to Def. 2. Similarly, in the time interval $[t_5, t_6)$, only $J(\mathcal{R}^6)$ is pi-blocked. Jobs pi-blocked due to Rule L3 are colored blue in Fig. 2.*

From Ex. 2, we see that Rule L3 ensures all jobs of unsatisfied requests are pi-blocked for an equal amount of time in each time interval $[t_i, t_{i+1})$ where $i \in [n-m, n)$.

Lemma 2. *Under global scheduling and Rule L3, in each time interval $[t_i, t_{i+1})$ where $i \in [n-m, n)$, $J(\mathcal{R}^j)$ where $j \in (i, n]$ is pi-blocked for $t_{i+1} - t_i$ time units.*

Proof. Under the conditions of the lemma, $j - i \in [1, m]$. Thus, by Rule L3, each job $J(\mathcal{R}^j)$ where $j \in (i, n]$ is one of the m highest-priority pending jobs in $[t_i, t_{i+1})$. Additionally, from Lem. 1, each $J(\mathcal{R}^j)$ where $j \in (i, n]$ is suspended, and thus unable to be scheduled in $[t_i, t_{i+1})$. Under global scheduling, $m = c$, therefore by Def. 2, each $J(\mathcal{R}^j)$ is pi-blocked for $t_{i+1} - t_i$ time units in each $[t_i, t_{i+1})$. □

> **L4.** For any integers $i \in [1, n-m)$ and $k \in [0, n-i)$, in each time interval
> $$I_{i,k} = \left[t_i + \frac{k(t_{i+1} - t_i)}{n-i}, t_i + \frac{(k+1)(t_{i+1} - t_i)}{n-i} \right),$$
> the h^{th} highest-priority pending job where $h \in [1, m]$ is job $J(\mathcal{R}^j)$, where $j = i + 1 + (h - 1 + km \bmod n - i)$.

Rule L4 ensures that all unsatisfied requests are pi-blocked for an equal amount of time in each time interval $[t_i, t_{i+1})$ where $i \in [1, n-m)$. This is done by dividing the interval $[t_i, t_{i+1})$ into $n-i$ sub-intervals $(I_{i,0}, I_{i,1}, ..., I_{i,n-i-1})$ of equal length. In each sub-interval $I_{i,k}$, there are m "slots" for jobs to become one of the m highest-priority pending jobs. The mod operator in Rule L4 ensures that the m jobs out of the set of jobs with unsatisfied requests *fill* these m slots cyclically. We formally define what it means to fill a slot below.

Definition 7. *In each time interval $[t_i, t_{i+1})$ where $i \in [1, n-m)$, we denote the slot for the h^{th} highest-priority pending job in $I_{i,k}$ as s_{h-1+km}. We say that $J(\mathcal{R}^j)$* fills *slot s_x in $I_{i,k}$ iff $j = i + 1 + (x \bmod n - i)$.*

We illustrate the behavior of Rule L4 and how slots are filled in the following example depicted in Fig. 3.

Example 3. *Consider the same task set as in Ex. 2. Thus, $n = 6$ and $c = m = 2$. Here we focus on the pi-blocking of $J(\mathcal{R}^2)$ to $J(\mathcal{R}^6)$ in the time interval $[t_1, t_2)$.*

According to Def. 7, when $i = 1$ and $k = 0$, the slot for the highest-priority job ($h = 1$) in $I_{1,0}$ is s_0. By Rule L4, $J(\mathcal{R}^2)$ fills s_0 and becomes the highest-priority job in $I_{1,0}$. This is because the equation for j in Rule L4 evaluates to 2 when $h = 1$. Meanwhile, the slot for the second highest-priority job ($h = 2$) in $I_{1,0}$ is s_1 and is filled by $J(\mathcal{R}^3)$. This means that under Def. 2, $J(\mathcal{R}^2)$ and $J(\mathcal{R}^3)$ are pi-blocked for $(t_2-t_1)/5$ time units in $I_{1,0}$. When $k = 1$, by Def. 7, s_2 and s_3 are slots for the $c = 2$ highest-priority pending jobs in $I_{1,1}$. These two slots are filled by $J(\mathcal{R}^4)$ and $J(\mathcal{R}^5)$ due to Rule L4, implying that $J(\mathcal{R}^4)$ and $J(\mathcal{R}^5)$ are pi-blocked for $(t_2 - t_1)/5$ time units in $I_{1,1}$. When $k = 2$, similarly, $J(\mathcal{R}^6)$ and $J(\mathcal{R}^2)$ fill slots s_4 and s_5 and are pi-blocked for $(t_2-t_1)/5$ time units in $I_{1,2}$. This pattern then continues according to Fig. 3 until t_2. We can see that in the time interval $[t_1, t_2)$, each job $J(\mathcal{R}^j)$ where $j > i$ (job of unsatisfied requests) is pi-blocked for a total of $2(t_2 - t_1)/5$ time units.

To prove that Rule L4 ensures all unsatisfied requests are pi-blocked equally in each time interval $[t_i, t_{i+1})$, we focus on the number of times a job $J(\mathcal{R}^j)$ fills a slot for one of the m highest-priority pending jobs in $[t_i, t_{i+1})$. We begin by analyzing the properties of slots in each $I_{i,k}$.

Definition 8. *For each interval $I_{i,k}$ where $i \in [1, n-m)$, let $S_k = \{s_{h-1+km} \mid h \in [1, m]\}$ be the set of slots for the m highest-priority pending jobs in $I_{i,k}$.*

Lemma 3. *For $i \in [1, n-m)$, a job $J(\mathcal{R}^j)$ fills at most one slot in each S_k.*

Proof. Suppose to the contrary and $J(\mathcal{R}^j)$ fills at least two slots in S_k, say, s_x and s_y where $x \neq y$. By Def. 7, $j = i + 1 + (x \bmod n - i)$ and $j = i + 1 + (y \bmod n - i)$. Since $x \neq y$, due to the mod operator, the difference between x and y must be at least $n - i$. However, by Def. 8, the indices of $s_x \in S_k$ and $s_y \in S_k$ can differ by at most m. Since the lemma assumes $i \in [1, n-m)$, we have $n-i > m$, so s_x and s_y cannot both be in S_k. Contradiction. □

Next, we show that a job $J(\mathcal{R}^j)$ filling a slot in S_k corresponds to $J(\mathcal{R}^j)$ being pi-blocked in $I_{i,k}$.

Lemma 4. *Under global scheduling, for $i \in [1, n-m)$, $j > i$, and $k \in [0, n-i)$, $J(\mathcal{R}^j)$ is pi-blocked in $I_{i,k}$ iff $J(\mathcal{R}^j)$ fills a slot in S_k.*

Proof. First we show that if $J(\mathcal{R}^j)$ fills a slot in S_k, then $J(\mathcal{R}^j)$ is pi-blocked in $I_{i,k}$. Since $J(\mathcal{R}^j)$ fills a slot in S_k, then by Defs. 7 and 8, $j = i+1+(h-1+km \bmod n-i)$ where $h \in [1, m]$. This is the exact condition for j in Rule L4 for $J(\mathcal{R}^j)$ to become the h^{th} highest-priority pending job in $I_{i,k}$. Thus, $J(\mathcal{R}^j)$ is among the m highest-priority pending jobs in $I_{i,k}$. Because the lemma assumes $j > i$, $J(\mathcal{R}^j)$ is suspended in $[t_i, t_{i+1})$ by Lem. 1. Since the lemma also assumes $k \in [0, n-i)$, by the definition of $I_{i,k}$ in Rule L4, $I_{i,k} \subset [t_i, t_{i+1})$, implying that $J(\mathcal{R}^j)$ is suspended in $I_{i,k}$. Therefore, since $J(\mathcal{R}^j)$ is suspended but is one of the $m = c$ highest-priority pending jobs in $I_{i,k}$, by Def. 2, $J(\mathcal{R}^j)$ is pi-blocked in $I_{i,k}$.

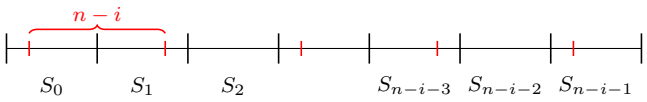


Fig. 4: A number line containing the set of slots for the m highest-priority pending jobs in $[t_i, t_{i+1})$. The indices of slots filled by a job $J(\mathcal{R}^j)$ are marked in red. Since for consecutive values of z, $f(z)$ differs by $n-i$, there are m values of z where $f(z) \in [0, m(n-i))$.

We now show that if $J(\mathcal{R}^j)$ is pi-blocked in $I_{i,k}$, then it fills a slot in S_k. If $J(\mathcal{R}^j)$ is pi-blocked in $I_{i,k}$, then by Def. 2, $J(\mathcal{R}^j)$ is one of the m highest-priority pending jobs in $I_{i,k}$. Thus, from Rule L4, $j = i+1+(h-1+km \bmod n-i)$ where $h \in [1, m]$. By Defs. 7 and 8, $J(\mathcal{R}^j)$ fills a slot in S_k. □

From Rule L4, the time interval $[t_i, t_{i+1})$ is composed of the sub-intervals $I_{i,k}$ where $k \in [0, n-i)$. Thus, with Lem. 3 and 4, we see that if each job fills a slot in an equal number of sets S_k where $k \in [0, n-i)$, then these jobs will be pi-blocked for an equal number of time intervals $I_{i,k}$ in $[t_i, t_{i+1})$.

Lemma 5. *For $i \in [1, n-m)$, a job $J(\mathcal{R}^j)$ where $j > i$ fills a total of m slots in $\bigcup_{k=0}^{n-i-1} S_k$.*

Proof. For notational brevity, let $S = \bigcup_{k=0}^{n-i-1} S_k$. Def. 8 implies that a slot s_x where $x \in [km, m+km)$ is in S_k. Thus, it can be verified that $S = \{s_x \mid x \in [0, (n-i)m)\}$. Fig. 4 illustrates the sets of slots comprising S on a number line. We now want to show that $J(\mathcal{R}^j)$ fills m slots in S. By Def. 7, $J(\mathcal{R}^j)$ fills slot s_x if $j = i + 1 + (x \bmod n - i)$. From the mod operator, $J(\mathcal{R}^j)$ fills slot s_x if there exists some integer z where $j-i-1+z(n-i) = x$. Letting $f(z) = j-i-1+z(n-i)$, we see that each value of z where $f(z) = x \in [0, (n-i)m)$ is a slot in S. Thus, we want to show that there are m values of z where $f(z) \in [0, (n-i)m)$. These values are illustrated in Fig. 4.

For consecutive values of z, $f(z)$ differs by $n - i$. Thus, the number of values of z where $f(z) \in [0, (n-i)m)$ is equivalent to the number of integers contained in the interval $[0, m)$. Since the number of integers contained in $[0, m)$ is m, there are m values of z where $f(z) \in [0, (n-i)m)$. □

Finally, the fact that all jobs of unsatisfied requests are pi-blocked for an equal amount in time interval $[t_i, t_{i+1})$ follows directly from Lems. 3 to 5.

Lemma 6. *Under global scheduling, in each time interval $[t_i, t_{i+1})$ where $i \in [1, n-m)$, $J(\mathcal{R}^j)$ where $j > i$ is pi-blocked for $\frac{m(t_{i+1}-t_i)}{n-i}$ time units.*

Proof. Lems. 3 and 5 imply that there are m values of $k \in [0, n-i)$ where $J(\mathcal{R}^j)$ fills a slot in S_k. By Lem. 4, $J(\mathcal{R}^j)$ where $j > i$ is pi-blocked in $I_{i,k}$ for m values of $k \in [0, n-i)$. From Rule L4, since each time interval $I_{i,k}$ where $k \in [0, n-i)$ is in $[t_i, t_{i+1})$ and is of length $\frac{t_{i+1}-t_i}{n-i}$, the lemma follows. □

B. Lower-Bound Proof

In this section, we show that for our pathological task set Γ, the priority changes of which are possible under non-JLFP

scheduling, some job must experience per-request pi-blocking of $\Omega(m + m(H_n - H_m))$ time units under the considered (arbitrary) locking protocol. As such, there exist situations where $\Omega(m + m(H_n - H_m))$ pi-blocking is unavoidable.

We begin by focusing on the pi-blocking of $J(\mathcal{R}^n)$.

Lemma 7. *$J(\mathcal{R}^n)$ is pi-blocked for*

$$\sum_{i=1}^{n-m-1} \frac{m(t_{i+1} - t_i)}{n-i} + \sum_{i=n-m}^{n-1} t_{i+1} - t_i \tag{1}$$

time units in the time interval $[t_1, t_n)$.

Proof. We prove the lemma by examining the amount $J(\mathcal{R}^n)$ is pi-blocked in the time intervals $[t_1, t_{n-m})$ and $[t_{n-m}, t_n)$. From Lem. 6, $J(\mathcal{R}^n)$ is pi-blocked for $\frac{m(t_{i+1}-t_i)}{n-i}$ time units in each time interval $[t_i, t_{i+1})$ where $i \in [1, n-m)$. Therefore, in $[t_1, t_{n-m})$, $J(\mathcal{R}^n)$ is pi-blocked for $\sum_{i=1}^{n-m-1} \frac{m(t_{i+1}-t_i)}{n-i}$ time units. After t_{n-m}, by Lem. 2, $J(\mathcal{R}^n)$ is pi-blocked for $t_{i+1} - t_i$ time units in each time interval $[t_i, t_{i+1})$ where $i \in [n-m, n)$. Thus, from t_{n-m} to t_n, $J(\mathcal{R}^n)$ is pi-blocked for $\sum_{i=n-m}^{n-1} t_{i+1} - t_i$ time units. □

Since we can obtain a lower bound for each $t_{i+1} - t_i$ term in (1) due to Prop. 2, we can use Lem. 7 to obtain the following pi-blocking lower bound under non-JLFP scheduling.

Theorem 1. *Under any mutex locking protocol, there exists a task set for which some job experience per-request pi-blocking of $\Omega(m + m(H_n - H_m))$ time units in a schedule allowed by non-JLFP scheduling.*

Proof. Consider the pathological task set Γ under global non-JLFP scheduling. By Def. 1, job priorities can change according to Rules L3 and L4 under non-JLFP scheduling. Thus, by Lem. 7 and Prop. 2, $J(\mathcal{R}^n)$ is pi-blocked for a minimum of

$$\sum_{i=1}^{n-m-1} \frac{m}{n-i} + \sum_{i=n-m}^{n-1} 1 \tag{2}$$

time units in the time interval $[t_1, t_n)$. By Def. 5, the first term of (2) is equal to $m(H_{n-1} - H_m)$. Additionally, the second term of (2) is equal to m. Therefore, $J(\mathcal{R}^n)$ is pi-blocked for at least $m + m(H_{n-1} - H_m)$ time units in $[t_1, t_n + 1)$. Since $n > m$, $H_n < 1$, which implies that $\Omega(m + m(H_n - H_m))$ lower-bounds the per-request pi-blocking of a job. □

IV. The NJLP

In this section, we present the *non-JLFP locking protocol (NJLP)*, a suspension-based mutex locking protocol that is asymptotically optimal under non-JLFP scheduling. We describe the protocol in Sec. IV-A, and establish its asymptotically optimality in Sec. IV-B.

A. The NJLP

To describe the operation of the NJLP, we first formally quantify the total time a job is pi-blocked due to one of its resource requests. We begin by considering the time instants when requests for a resource ℓ become satisfied.

Definition 9. *Let $\mathcal{R}^i$ be the i^{th} request for resource ℓ to be satisfied. We denote the time $\mathcal{R}^i$ becomes satisfied as the satisfaction point t_i.*

Accumulated pi-blocking. Between two satisfaction points t_i and t_{i+1}, the job of any unsatisfied request $\mathcal{R}$ for ℓ is suspended, and therefore can be pi-blocked under Def. 2.

Definition 10. *Let $\Delta_i(\mathcal{R})$ be the pi-blocking duration incurred by a request $\mathcal{R}$ in the interval $[t_i, t_{i+1})$.*

From the time $J(\mathcal{R})$ issues $\mathcal{R}$ until the time $\mathcal{R}$ becomes satisfied, $J(\mathcal{R})$ can accumulate pi-blocking due to $\mathcal{R}$. We denote the pi-blocking accumulated by $J(\mathcal{R})$ due to $\mathcal{R}$ from its issuance to some satisfaction point t_i as the *accumulated pi-blocking* of $J(\mathcal{R})$ at t_i. Formally, we have the following.

Definition 11. *For a request $\mathcal{R}$ issued in $[t_i, t_{i+1})$, let $E(\mathcal{R}) = i$. The Accumulated pi-blocking (APB) of $J(\mathcal{R})$ at satisfaction point t_k is given by $B_k(\mathcal{R})$ where $B_k(\mathcal{R}) = \sum_{j=E(\mathcal{R})}^{k-1} \Delta_j(\mathcal{R})$ if $k \geq E(\mathcal{R}) + 1$ and $B_k(\mathcal{R}) = 0$ otherwise.*

NJLP rules. For each resource, ℓ, the NJLP consists of a single priority queue PQ_ℓ of unsatisfied requests. At each satisfaction point t_i, PQ_ℓ ensures that requests are ordered by the APB of their jobs with the head of PQ_ℓ having the job with the highest APB. The rules of the NJLP are as follows.

> **R1.** When a job $J_{i,j}$ issues a request $\mathcal{R}$ for resource ℓ at time t, if there are no active requests for ℓ at t, then $\mathcal{R}$ immediately becomes satisfied. Otherwise, $\mathcal{R}$ is added to PQ_ℓ. All jobs with requests in PQ_ℓ are suspended.
>
> **R2.** When a request for ℓ completes, the request at the head of PQ_ℓ, $\mathcal{R}$, is removed from PQ_ℓ and becomes satisfied, and $J(\mathcal{R})$ is resumed.

Since a request $\mathcal{R}$ completes when $J(\mathcal{R})$ executes for at most the request length $L(\mathcal{R})$ after $\mathcal{R}$ becomes satisfied, it is evident that Rule R2 ensures the following property.

Property 3. *$J(\mathcal{R}^i)$ can execute for at most $L(\mathcal{R}^i)$ time units in $[t_i, t_{i+1})$.*

Progress mechanism. To ensure that jobs of satisfied requests can make progress, and to ensure that jobs experience bounded pi-blocking, we use the migratory priority inheritance progress mechanism alongside the NJLP. However, under global scheduling, priority inheritance can be used as a substitute to reduce overheads. Since both mechanisms ensure Prop. 1, and the set of requests waiting for the completion of the satisfied request is PQ_ℓ, we have the following corollary of Prop. 1.

Corollary 1. *At time t, if request $\mathcal{R}$ is satisfied, and there exists a request $\mathcal{R}' \in PQ_\ell$ where $J(\mathcal{R}')$ is pi-blocked, then $J(\mathcal{R})$ is scheduled.*

Using Cor. 1, we can show that the progress mechanism ensures that between successive satisfaction points, an unsatisfied request can be pi-blocked for at most L_ℓ time units.

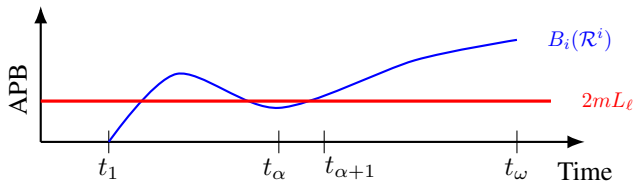


Fig. 5: Plot of $B_i(\mathcal{R}^i)$ for $i \in [1, \omega)$. The red line indicates the point where $B_i(\mathcal{R}^i)$ reaches $2mL_\ell$. Note that $B_1(\mathcal{R}^1) = 0$ as $\mathcal{R}^1$ is the first request to be satisfied for resource ℓ. We see that after t_α, $B_i(\mathcal{R}^i) \geq 2mL_\ell$.

Lemma 8. *For each request $\mathcal{R}$ for resource ℓ and any satisfaction point t_i, $\Delta_i(\mathcal{R}) \leq L_\ell$.*

Proof. Suppose not, and $\Delta_i(\mathcal{R}) > L_\ell$, then by Def. 10, $J(\mathcal{R})$ is pi-blocked for more than L_ℓ time units in $[t_i, t_{i+1})$. Since $J(\mathcal{R})$ is pi-blocked, Def. 2 implies that it is suspended. When $J(\mathcal{R})$ becomes suspended, $\mathcal{R}$ is added to PQ_ℓ by Rule R1. Therefore, by Cor. 1, $J(\mathcal{R}^i)$ is scheduled for more than L_ℓ time units in $[t_i, t_{i+1})$. Since L_ℓ is the maximum length of any request for ℓ, we have a contradiction with Prop. 3. □

B. Maximum Pi-blocking Under NJLP

To derive the maximum pi-blocking under the NJLP, we focus on the worst-case pi-blocking experienced by a job due to a request of interest $\mathcal{R}^*$. In our analysis, we let t_ω denote the satisfaction point where $\mathcal{R}^*$ becomes satisfied. Since $J(\mathcal{R}^*)$ only suspends until $\mathcal{R}^*$ becomes satisfied at t_ω, the pi-blocking of $J(\mathcal{R}^*)$ due to $\mathcal{R}^*$ is maximal at t_ω. To compute the maximum pi-blocking of $J(\mathcal{R}^*)$ at t_ω, we begin by examining the pi-blocking of $J(\mathcal{R}^*)$ at some earlier satisfaction point where the pi-blocking of $J(\mathcal{R}^*)$ can be bounded.

Definition 12. *Let t_α be the last satisfaction point at or before t_ω where $B_\alpha(\mathcal{R}^\alpha) < 2mL_\ell$.*

Def. 12 is defined such that at t_α, the job of the satisfied request has an APB less than $2mL_\ell$. A plot of $B_i(\mathcal{R}^i)$ for each t_i from t_1 to the point when $\mathcal{R}^*$ becomes satisfied is illustrated in Fig. 5. From Fig. 5, we see that if $B_\omega(\mathcal{R}^\omega) > 2mL_\ell$, then $t_\alpha < t_\omega$, and $t_\alpha = t_\omega$ otherwise. From Def. 12, we can see that the APB of $J(\mathcal{R}^*)$ is less than $2mL_\ell$ at t_α. This is because if $B_\alpha(\mathcal{R}^*) > 2mL_\ell$ (can only occur if $\mathcal{R}^*$ is issued before t_α by Def. 11), then $B_\alpha(\mathcal{R}^\alpha) > 2mL_\ell$ due to Rule R2. Using this, if we can upper-bound the time difference between t_ω and t_α, we can also upper-bound the APB of $J(\mathcal{R}^*)$ at t_ω.

Properties of total APB. We can upper-bound the time difference between t_ω and t_α by examining the total APB of all jobs with requests in PQ_ℓ and how it changes over time.

Definition 13. *For each satisfaction point t_i, let Q_i be the set of requests in PQ_ℓ at t_i.*

Since there can be at most n pending jobs, and each job has at most one incomplete request, we have the following.

Property 4. *At each selection point t_i, $|Q_i| < n$.*

Using Def. 13, we formally define the total APB as follows.

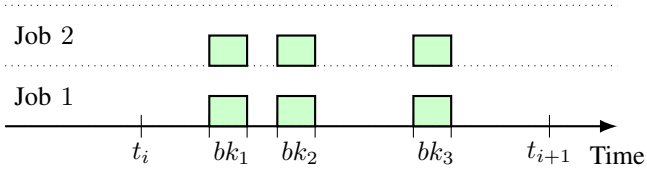


Fig. 6: Example timeline of $[t_i, t_{i+1})$ on a systems with $m = 2$ processors. At each time instant, at most m jobs can be among the c highest-priority pending jobs in their cluster. Green indicates times when a job whose request is in S is one of those $m = 2$ jobs (labeled Job 1 and Job 2). Since the total duration of all bk_j intervals is at most L_ℓ, jobs with requests in S are pi-blocked for at most mL_ℓ time units in $[t_i, t_{i+1})$.

Definition 14. *For each satisfaction point t_i, let $TB_i = \sum_{\mathcal{R} \in Q_i} B_i(\mathcal{R})$ denote the total APB of all jobs whose requests are in PQ_ℓ at t_i.*

From Def. 11 and 14, we can see that changes in the total APB are dependent on the pi-blocking experienced by jobs between satisfaction points. Therefore, to examine the change in total APB, we first show the following.

Lemma 9. *For any set S of requests for resource ℓ and satisfaction point t_i, $\sum_{\mathcal{R} \in S} \Delta_i(\mathcal{R}) \leq mL_\ell$.*

Proof. Let $bk = \{bk_1, bk_2, ...\}$ be the set of time intervals in $[t_i, t_{i+1})$ when a job whose request is in S is pi-blocked. We additionally let $|bk_j|$ denote the duration of bk_j. Each bk_j is marked on the timeline in Fig. 6. Since a job $J(\mathcal{R})$ where $\mathcal{R} \in S$ is pi-blocked in each bk_j, by Def. 2, $J(\mathcal{R})$ is suspended, and therefore $\mathcal{R}$ is in PQ_ℓ due to Rule R1. Therefore, by Cor. 1, $J(\mathcal{R}^i)$ is scheduled for $|bk_j|$ time units in bk_j. By Prop. 3, $J(\mathcal{R}^i)$ cannot be scheduled for more than L_ℓ time units in $[t_i, t_{i+1})$. Thus, $\sum_{bk_j \in bk} |bk_j| \leq L_\ell$. Since there are m/c clusters and each cluster can have at most c jobs with the c highest priorities, at most m jobs can be among the c highest-priority pending jobs in their cluster at any time. Therefore, by Def. 2, in each bk_j, jobs are pi-blocked for a total of at most $m|bk_j|$ time units. Fig. 6 illustrates this. Thus, in $[t_i, t_{i+1})$, jobs with a request in S are pi-blocked for a total of at most $\sum_{bk_j \in bk} m|bk_j| \leq mL_\ell$ time units. The lemma follows from Def. 10. □

Using Lem. 9, we can upper-bound the total APB change across subsequent satisfaction points.

Lemma 10. *For each pair of satisfaction points t_{i-1} and t_i, $TB_i - TB_{i-1} \leq mL_\ell - B_i(\mathcal{R}^i)$*

Proof. At t_i, by Def. 9, $\mathcal{R}^i$ becomes satisfied, and by Rule R2, is removed from PQ_ℓ. Additionally, by Rule R1, all requests issued in $[t_{i-1}, t_i)$ are in PQ_ℓ at t_i. Therefore, $Q_i = Q_{i-1} \cup S - \mathcal{R}^i$ where S is the set of requests issued in $[t_{i-1}, t_i)$. Hence, by Def. 14, we have $TB_i - TB_{i-1} =$

$$\left(\sum_{\mathcal{R} \in Q_{i-1}} B_i(\mathcal{R}) - B_{i-1}(\mathcal{R})\right) + \left(\sum_{\mathcal{R} \in S} B_i(\mathcal{R})\right) - B_i(\mathcal{R}^i). \tag{3}$$

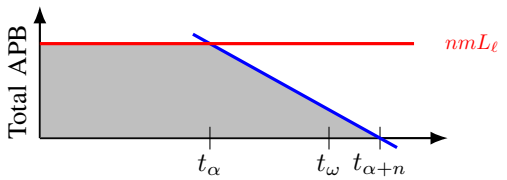


Fig. 7: Plot of the possible total APB at satisfaction points. The red line indicates the upper bound on the total APB in Lem. 11. The slope of the blue line indicates the rate of decrease for total APB after t_α, given in Lem. 10. The region where total APB values are valid is shaded.

From Def. 11, each $B_i(\mathcal{R}) - B_{i-1}(\mathcal{R})$ term in the first summation equals $\Delta_{i-1}(\mathcal{R})$. Also, since all requests in S are issued at or after t_{i-1}, by Def. 11, each term in the second summation equals $\Delta_{i-1}(\mathcal{R})$. We therefore have $TB_i - TB_{i-1} =$

$$\left(\sum_{\mathcal{R}\in Q_{i-1}\cup S} \Delta_{i-1}(\mathcal{R})\right) - B_i(\mathcal{R}^i). \tag{4}$$

By Lem. 9, the first summation is at most mL_ℓ. Therefore, we have $TB_i - TB_{i-1} \leq mL_\ell - B_i(\mathcal{R}^i)$. □

Due to Lem. 10, we can also upper-bound the total APB.

Lemma 11. $TB_i \leq nmL_\ell$.

Proof. Suppose to the contrary, and let t_x be the first satisfaction point where $TB_x > nmL_\ell$.[4] Since $TB_{x-1} \leq nmL_\ell$, we have $TB_x > TB_{x-1}$, which by Lem. 10 implies

$$B_x(R^x) < mL_\ell. \tag{5}$$

However, by Prop. 4, $|Q_x| < n$. Thus, since $TB_x > nmL_\ell$, by Def. 14 and the pigeonhole principle, at least one request $\mathcal{R} \in Q_x$ satisfies $B_x(\mathcal{R}) \geq mL_\ell$. Since Rule R2 selects the request with the highest APB to become satisfied, $B_x(\mathcal{R}^x) \geq B_x(\mathcal{R}) \geq mL_\ell$. This contradicts (5) □

Using the conditions for the total APB established in Lems. 10 and 11, we can determine the latest possible time of t_ω in the following lemma, whose logic is depicted by Fig. 7.

Lemma 12. *$\mathcal{R}^*$ is, at the latest, the n^{th} request to be satisfied after t_α. In other words, $\omega - \alpha \leq n$.*

Proof. By Def. 12, for each t_i where $i \in (\alpha, \omega]$, $B_i(\mathcal{R}^i) \geq 2mL_\ell$. Thus, by Lem. 10, for $i \in (\alpha, \omega]$, $TB_i - TB_{i-1} \leq -mL_\ell$. Now consider

$$\sum_{i=\alpha+1}^{\omega} TB_i - TB_{i-1}. \tag{6}$$

[4]This is only possible when $x > 1$. This is because, due to Def. 11, for a request $\mathcal{R}$, $B_x(\mathcal{R}) > 0$ only when $x > E(\mathcal{R})$ and $E(\mathcal{R}) \geq 1$.

Since each $TB_i - TB_{i-1}$ is at most $-mL_\ell$, (6) is at most $-(\omega-\alpha)mL_\ell$. Additionally, by cancelling out like terms, (6) equals $TB_\omega - TB_\alpha$. Therefore,

$$\begin{aligned} TB_\omega - TB_\alpha &\leq -(\omega-\alpha)mL_\ell \\ TB_\omega &\leq nmL_\ell - (\omega-\alpha)mL_\ell \quad \text{(By Lem. 11)} \\ TB_\omega &\leq (n-(\omega-\alpha))mL_\ell. \end{aligned}$$

Now suppose the lemma is false, and $\omega - \alpha > n$. This would imply that $TB_\omega < 0$. Since Defs. 10, 11, and 14, imply that $TB_\omega \geq 0$, we have a contradiction. □

Determining maximum pi-blocking. Having upper-bounded the latest time that $\mathcal{R}^*$ can become satisfied in Lem. 12, we proceed to determine the maximum pi-blocking of $J(\mathcal{R}^*)$ due to $\mathcal{R}^*$. We begin by examining the last requests to become satisfied before $\mathcal{R}^*$.

Definition 15. *Let Ω_i denote the set of requests that become satisfied at each satisfaction point t_j where $j \in [i, \omega]$.*

Notice that by Def. 15, $\mathcal{R}^*$ is the only element in Ω_ω. This implies that at t_ω, the average APB of jobs for all requests in Ω_ω is equal to the APB of $J(\mathcal{R}^*)$ at t_ω. Therefore, to determine the maximum pi-blocking of $J(\mathcal{R}^*)$, we examine the increase in average APB from t_α to t_ω.

Definition 16. *At t_i, the average APB of jobs whose requests are in Ω_i is given by $AVG_i = \sum_{\mathcal{R}\in\Omega_i} B_i(\mathcal{R})/|\Omega_i|$.*

Lemma 13. *At satisfaction point t_α, $AVG_\alpha < 2mL_\ell$.*

Proof. Suppose to the contrary that $AVG_\alpha \geq 2mL_\ell$. Then, there exists a request $\mathcal{R} \in \Omega_\alpha$ where $B_\alpha(\mathcal{R}) \geq 2mL_\ell$. $B_\alpha(\mathcal{R}) \geq 2mL_\ell$ implies by Def. 11 that $\mathcal{R}$ is issued, and therefore enters PQ_ℓ by Rule R1 before t_α. Since Rule R2 selects the request with the highest APB at t_α to become satisfied, $B_\alpha(\mathcal{R}^\alpha) \geq B_\alpha(\mathcal{R}) \geq 2mL_\ell$. However, since Def. 12 implies $B_\alpha(\mathcal{R}^\alpha) < 2mL_\ell$, we have a contradiction. □

Lemma 14. *At satisfaction point t_ω, $AVG_\omega < L_\ell(2m + \sum_{i=\alpha+1}^{\omega} \min(1, \frac{m}{\omega-i+1}))$.*

Proof. Consider $X = \sum_{i=\alpha+1}^{\omega}(AVG_i - AVG_{i-1})$. By expanding out the summation and cancelling like terms, $X = AVG_\omega - AVG_\alpha$. From Lem. 13, $AVG_\alpha < 2mL_\ell$, so

$$AVG_\omega < 2mL_\ell + X. \tag{7}$$

To determine the value of X, we first examine $AVG_i - AVG_{i-1}$ for $i \in (\alpha, \omega]$. From Def. 16, we have $AVG_i - AVG_{i-1} =$

$$\sum_{\mathcal{R}\in\Omega_i} B_i(\mathcal{R})/|\Omega_i| - \sum_{\mathcal{R}\in\Omega_{i-1}} B_{i-1}(\mathcal{R})/|\Omega_{i-1}|.$$

From Def. 15, we have $\Omega_{i-1} = \Omega_i \cup \{\mathcal{R}^{i-1}\}$. This also implies that $|\Omega_i| < |\Omega_{i-1}|$. Thus, $AVG_i - AVG_{i-1} <$

$$\frac{1}{|\Omega_i|}\left(\left(\sum_{\mathcal{R}\in\Omega_i} B_i(\mathcal{R})\right) - \left(B_{i-1}(\mathcal{R}^{i-1}) + \sum_{\mathcal{R}\in\Omega_i} B_{i-1}(\mathcal{R})\right)\right).$$

By Def. 11, we can see that subtracting the second summation from the first results in $\sum_{\mathcal{R}\in\Omega_i}\Delta_i(\mathcal{R})$, which by Lem. 9 is at most mL_ℓ. This implies that $AVG_i - AVG_{i-1} < mL_\ell/|\Omega_i|$. However, due to Lem. 8, $\Delta_i(\mathcal{R}) \leq L_\ell$ for each $\mathcal{R} \in \Omega_i$. This implies that $AVG_i - AVG_{i-1} \leq L_\ell$. Therefore, each $AVG_i - AVG_{i-1} \leq \min(1, m/|\Omega_i|)L_\ell$, which implies that

$$X \leq \sum_{i\in\alpha+1}^{\omega} \min(1, m/|\Omega_i|)L_\ell. \tag{8}$$

By substituting (8) into (7), we have $AVG_\omega < L_\ell(2m + \sum_{i\in\alpha+1}^{\omega}\min(1, m/|\Omega_i|))$. Since Def. 15 implies that $|\Omega_i| = \omega - i + 1$, the lemma follows. □

Since $J(\mathcal{R}^*)$ resumes from suspension by Rule R2 after $\mathcal{R}^*$ is satisfied at t_ω, $J(\mathcal{R}^*)$ cannot be pi-blocked due to $\mathcal{R}^*$ after t_ω. Therefore, the upper-bound of AVG_ω in Lem. 14, can be used to compute the maximum pi-blocking under NJLP.

Theorem 2. *Under the NJLP, $J(\mathcal{R}^*)$ is pi-blocked for less than $(3m - 2 + m(H_n - H_{m-1}))L_\ell$ time units due to $\mathcal{R}^*$.*

Proof. By Def. 15, $\Omega_\omega = \{\mathcal{R}^*\}$, implying by Def. 16 that $AVG_\omega = B_\omega(\mathcal{R}^*)$. By Lem. 14, $B_\omega(\mathcal{R}^*) < L_\ell(2m + \sum_{i=\alpha+1}^{\omega}\min(1, \frac{m}{\omega-i+1}))$. Since $\omega - \alpha \leq n$ due to Lem. 12, we have $B_\omega(\mathcal{R}^*) < L_\ell(2m + \sum_{i=0}^{n-1}\min(1, \frac{m}{n-i}))$, hence

$$B_\omega(\mathcal{R}^*) < L_\ell\left(2m + \sum_{i=0}^{n-m}\frac{m}{n-i} + \sum_{i=n-m+1}^{n-1} 1\right).$$

The first summation is equivalent to $m(H_n - H_{m-1})$, and the second is equivalent to $m - 1$. Therefore $B_\omega(\mathcal{R}^*) < (3m - 1 + m(H_n - H_{m-1}))L_\ell$. Thus, $J(\mathcal{R}^*)$ is pi-blocked for less than $(3m-1+m(H_n-H_{m-1}))L_\ell$ time units due to $\mathcal{R}^*$. □

From Thm. 2, we see that under the NJLP, each request of a job causes less than $(3m - 1 + m(H_n - H_{m-1}))L_\ell$ time units of pi-blocking. Additionally, since $(3m - 1 + m(H_n - H_{m-1}))L_\ell = O(m + m(H_n - H_m))$ (note that $mH_m = mH_{m-1} + 1$), the NJLP is asymptotically optimal, by Thm. 1.

Theorem 3. *Pi-blocking under the NJLP is asymptotically optimal under non-JLFP scheduling.*

Proof. Directly follows from Thm. 1 and 2. □

V. Implementation and Evaluation

To evaluate the performance of the NJLP, we implemented it using the priority-inheritance progress mechanism for the global EDZL scheduler in LITMUS$^{\mathrm{RT}}$ [11], a real-time extension of the Linux kernel.

NJLP implementation. Unlike other locking protocols such as [7]–[9], where pi-blocking is merely of analytical interest, under the NJLP, the pi-blocking of each job must be tracked on-the-fly to correctly order requests in PQ_ℓ. Apart from tracking pi-blocking, Rules R1 and R2 in the NJLP can be trivially implemented. From the definition of pi-blocking in Def. 2, keeping track of each job's pi-blocking under global EDZL requires the OS to keep track of the m highest-priority *pending* jobs. This is similar to the global EDZL scheduling logic, which keeps track of the m highest-priority *ready* jobs in order to assign them to the m available processors.

Duplicate scheduling logic. While the EDZL scheduling logic closely resembles our desired mechanism to track pi-blocking, it cannot be used to keep track of both the m highest-priority pending jobs and the m highest-priority ready jobs, as these two job sets can be different.

Example 4. *Consider the set of pending jobs $\{J_1, J_2, J_3, J_4\}$ at time t on a two-processor system. Suppose that higher-indexed jobs have a higher priority at t. Then, the m highest-priority pending jobs at t are J_3 and J_4. Now suppose both J_3 and J_4 have an unsatisfied request at t. Then, by Rule R1, both J_3 and J_4 are suspended and therefore not ready. As a result, the m highest-priority ready jobs at t are J_1 and J_2.*

Due to scenarios like Ex. 4, we require two similar copies of the scheduling logic, one to keep track of the m highest-priority pending jobs, and the other to track the m highest-priority ready jobs. As a result, tracking job pi-blocking on-the-fly can double the overhead of the scheduling logic.

Non-optimal non-JLFP locking alternative. Since an implementation of the NJLP requires the OS to track pi-blocking, the protocol may incur increased overheads. Thus, in certain scenarios, a lower-overhead locking protocol with sub-optimal pi-blocking under non-JLFP scheduling may be preferable to the NJLP when considering the schedulability impacts of both pi-blocking and overheads together. One such non-optimal locking protocol is the (long) FMLP [4], which orders unsatisfied requests using a FIFO queue of size n and uses priority inheritance as a progress mechanism. Worst-case per-request pi-blocking under the FMLP is $\Omega(n)$ request lengths, which is sub-optimal. However, since the FMLP does not need to track pi-blocking, it can exhibit lower overheads compared to the NJLP. These properties make the FMLP an ideal benchmark for our evaluation of the NJLP.

In the rest of this section, we first compare the theoretical performance of the NJLP versus the FMLP in Sec. V-A. We then evaluate the overheads of LITMUS$^{\mathrm{RT}}$ implementations of both protocols in Sec. V-B. Finally, we use our overhead results in an overhead-aware schedulability study in Sec. V-C.

A. Theoretical Pi-Blocking Evaluation

In our first experimental evaluation, we compared the theoretical pi-blocking bound of the NJLP versus the FMLP without considering overheads. This was done by examining the pi-blocking bound of the NJLP (in Thm. 2) and the FMLP (in [4]) across task systems with $n \in [0, 64]$ scheduled on systems with $m \in \{1, 2, 4, 8\}$ processors. In this experiment, each job makes one request for resource ℓ. Since the maximum pi-blocking of both protocols are multiples of the longest per-resource request length, L_ℓ, we set $L_\ell = 1$. Our results are shown in Fig. 8, which shows the difference between the pi-

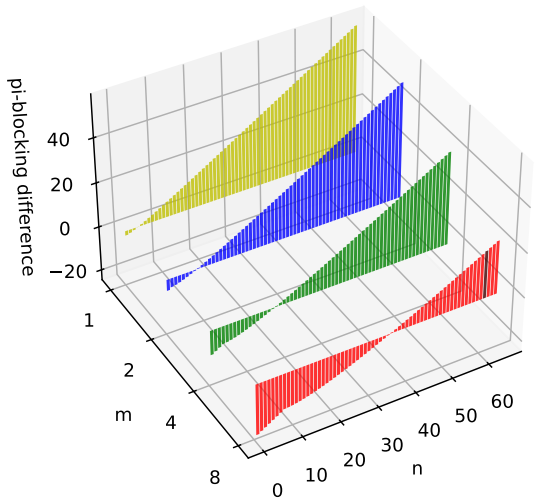


Fig. 8: Plot of the pi-blocking difference obtained from subtracting the maximum pi-blocking due to the NJLP from the maximum pi-blocking due to the FMLP. Red corresponds to the difference when $m = 8$, green corresponds to $m = 4$, blue corresponds to $m = 2$, and yellow corresponds to $m = 1$. For example, the darkened red bar at $n = 60$ and $m = 8$ (with a height of around 20) indicates that maximum pi-blocking is around 20 time units more under the FMLP than under the NJLP when $n = 60$ and $m = 8$.

blocking bound due to the NJLP versus the FMLP. From this figure, we can observe the following.

Observation 1. *When n is greater than approximately $4m$, the NJLP resulted in lower maximum pi-blocking than the FMLP.*

Obs. 1 shows that in systems with a moderate to high task count n, the NJLP performs significantly better than the FMLP in theory. This observation is unsurprising if we examine the pi-blocking bounds of the two protocols. The pi-blocking bound of the NJLP, given by Thm. 2 is $O(m+m(H_n-H_m))$. Since the n^{th} harmonic number can be approximated by $\ln(n)$, the pi-blocking bound of the NJLP increases logarithmically with n. On the other hand, the pi-blocking bound of the FMLP, $O(n)$, increases linearly with n. Clearly, as the number of tasks increases, maximum pi-blocking under the FMLP will increase faster than under the NJLP.

B. Overhead Evaluation

In Sec. V-A, we demonstrated that the NJLP theoretically outperforms the FMLP under high resource contention. However, since implementing the NJLP necessitates tracking job pi-blocking, the overhead of scheduling and releasing jobs will inevitably increase. Therefore, in this subsection, we examine **(i)** how tracking pi-blocking in the global EDZL scheduler affects job release and scheduling overheads, and **(ii)** compare the lock and unlock overheads in our implementation of the NJLP and the FMLP.

We conducted our overhead evaluations using synthetic task sets on a six-core 2.2 GHz Intel i7-8750H processor running LITMUS$^{\mathrm{RT}}$. To ensure accurate and consistent measurements, we disabled simultaneous multi-threading and dynamic power and frequency scaling on the processor.

Task-set generation. To evaluate overheads across different task-set sizes, we used Emberson et al.'s method [14] to generate task sets consisting of $n = \{10, 20, ..., 100\}$ tasks with a total utilization of 0.2. In each task set, the period of each task was randomly selected from $[10, 100]ms$. Additionally, all tasks were configured to share a single resource ℓ, with each job of each task τ_i issuing $N_{i,\ell} = 1$ request with a request length equal to $1/10^{th}$ of the task's WCET, C_i. All tasks in the generated task sets thrash both the cache and memory (as described next) while executing.

Interference. To obtain overhead data under cache and memory interference, background tasks were created for each core. Each background task and task in the generated task sets accesses random memory pages in a tight loop. This causes new memory pages to continuously cycle in and out of the cache, thereby thrashing both the cache and memory.

For task sets of equal size, we recorded the maximum observed overhead across 5 minutes of continuous execution, where we measured over 100,000 samples for each type of overhead (i.e., scheduling, release, lock, unlock). These measurements are shown in Fig. 9. From this figure, we make the following observation.

Observation 2. *Scheduling overheads due to tracking pi-blocking were at most 1.5 times the overhead without tracking pi-blocking. Additionally, lock/unlock overheads of the NJLP were also at most 1.5 times the overheads of the FMLP.*

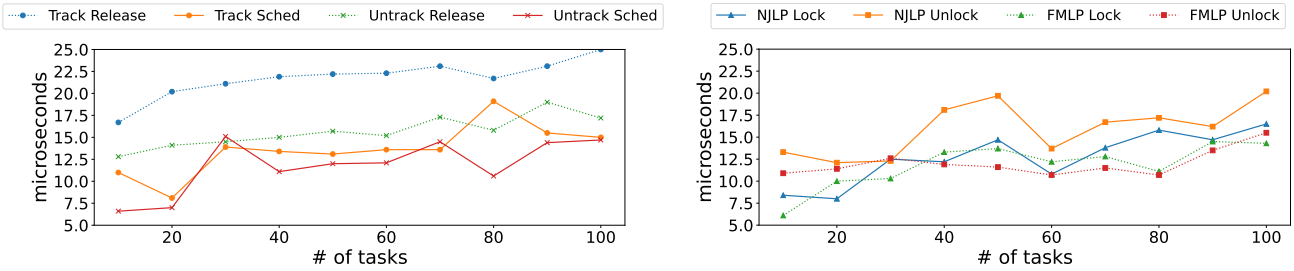
Obs. 2 indicates that while the overheads of using the NJLP are larger than when using the FMLP, this difference is not excessively large for our experimental setup. This indicates that in certain scenarios, it may be possible for the lower pi-blocking of the NJLP, demonstrated in Sec. V-A, to offset its increased overhead.

C. Schedulability Study

To assess the practical viability of the NJLP, we conducted an overhead-aware schedulability study using the overhead measurements from Sec. V-B. Since we only have the overhead measurements for a six-processor system, we only considered systems with $m = 6$ in our schedulability study.

Taskset generation. We used a similar method of generating task sets as prior locking-related schedulability studies [1], [5], [6], [22]. Task sets were randomly generated with the number of tasks n randomly selected from $[2m, 25]$ (small), $[4m, 50]$ (medium), or $[8m, 100]$ (large), and *normalized utilization*, $\sum_{i=1}^{n} \frac{C_i}{T_i \cdot m}$, in $\{0.2, 0.3, ..., 0.9\}$. Each task's utilization was generated using the method from [14], and its period is selected randomly from $[3, 33]ms$ (short), $[10, 100]ms$ (moderate), or $[50, 500]ms$ (long). Each task was defined to have an implicit deadline ($D_i = T_i$), and its execution cost was obtained by multiplying its utilization and period.

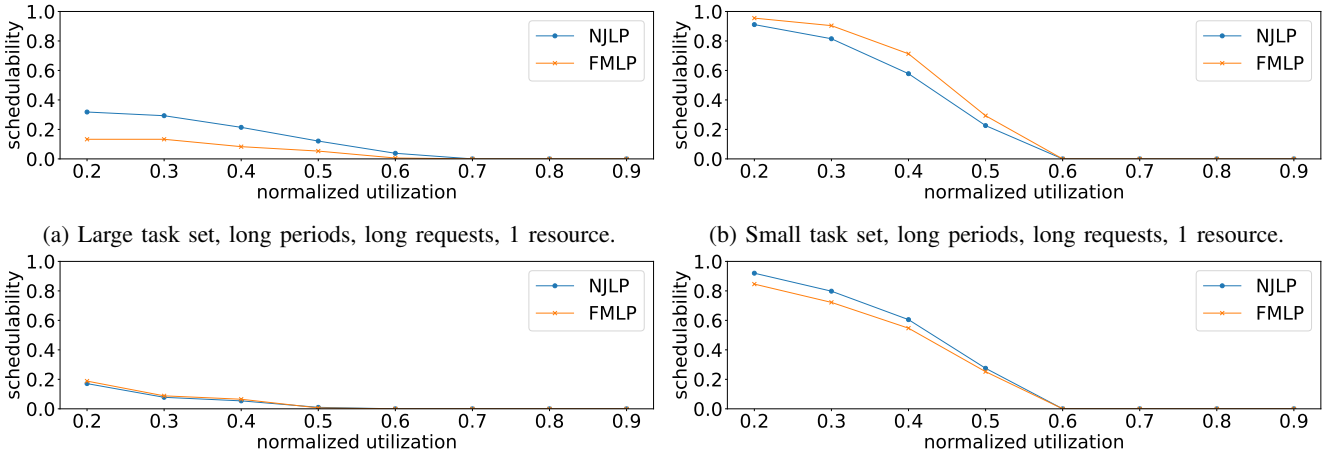
We considered systems with $\{1, 2, 3\}$ number of shared resources. Each task τ_i in a generated task set was configured



(a) Relevant scheduling overheads.

(b) Lock and unlock overheads.

Fig. 9: Comparison of the maximum observed overheads under the NJLP vs. the FMLP. Inset (a) compares the scheduling and job release overheads of "Track," a version of the global EDZL instrumented to track pi-blocking, and "Untrack," a version of it that does no such tracking. Inset (b) compares the lock/unlock overheads of the NJLP and the FMLP. We see that, counterintuitively, overheads sometimes *decrease* with increasing task set sizes. Originally noted in [5], this often occurs as the scheduler is invoked more frequently due to a larger number of tasks, causing kernel data structures to more likely remain cached between invocations.



(a) Large task set, long periods, long requests, 1 resource.

(b) Small task set, long periods, long requests, 1 resource.

(c) Medium task set, short periods, short requests, 1 resource.

(d) Medium task set, moderate periods, short requests, 1 resource.

Fig. 10: Results of our overhead-aware schedulability study.

to access a resource ℓ with probability $p = 0.5$. If τ_i can access ℓ, it does so through $N_{i,\ell} \in \{1, 2\}$ requests. We selected the maximum request length of each resource, L_ℓ, from two uniform distributions: $[1, 100]\mu s$ (short) and $[5, 1280]\mu s$ (long). We denote each combination of task set size, normalized utilization, task period, and request length, as a *scenario*, and generated 1,000 task systems for each scenario.

Schedulability test. In our overhead-aware schedulability study, we accounted for overheads by inflating task WCETs by the worst-observed release, schedule, and context-switch time according to the methods in [5]. Pi-blocking due to both the NJLP and the FMLP was also accounted for by inflating task WCETs (as in s-oblivious analysis) by the worst-observed lock and unlock overhead. Using the inflated task WCETs, we assessed the schedulability of each task set under global EDZL using the utilization test in [17]. For each scenario, we assessed acceptance ratios, which give the percentage of task systems that were schedulable under each locking protocol. We present a representative selection of our results in Fig. 10 and observe the following. The totality of our results, containing 54 graphs, can be found in the appendix [21].

Observation 3. *The NJLP performed better for large task sets, and the FMLP performed better for smaller task sets.*

From Obs. 3 we can conclude that despite the higher overheads in the NJLP implementation, its lower pi-blocking allows a higher percentage of larger task sets to be scheduled compared to the FMLP. However, for tasks with shorter periods, scheduling events can occur more often, amplifying the negative effects of overheads on schedulability. This leads to the following, which is supported by Figs. 10c and 10d.

Observation 4. *The NJLP performed better for tasks with larger periods, and the FMLP performed better for tasks with shorter periods.*

Due to space constraints, a full experimental evaluation of the NJLP that considers non-JLFP scheduling in contexts other than EDZL, the impacts of clustered scheduling, etc., is beyond the scope of this paper. Nonetheless, the results in this section do show that the issue of pi-blocking optimality under non-JLFP scheduling is important to consider not only for theoretical reasons, but for its practical implications.

VI. Conclusion

In this paper, we presented an $\Omega(m + m(H_n - H_m))$ pi-blocking lower bound that is applicable to any suspension-based locking protocol for non-JLFP-scheduled systems analyzed using s-oblivious techniques. This result shows that *it is pointless to try to design a generally applicable mutex locking protocol for such systems with asymptotically better pi-blocking because no such protocol exists*. We further showed that this lower bound is asymptotically tight by presenting the NJLP, the first asymptotically optimal mutex locking protocol for non-JLFP scheduling. Additionally, we presented experiments that suggest that pi-blocking optimality in non-JLFP-scheduled systems is an issue that is not merely of theoretical interest only. The paper opens up numerous avenues for further research pertaining to non-JLFP scheduled systems, including the design of locking protocols with overheads lower than the NJLP, protocols for specific non-JLFP schedulers (e.g., ones that change job priorities less frequently than in our lower-bound proof), protocols that allow lock nesting, protocols for k-exclusion and reader/writer synchronization, etc.

References

[1] S. Ahmed and J. H. Anderson, "Optimal Multiprocessor Locking Protocols Under FIFO Scheduling," in *ECRTS*, ser. Leibniz International Proceedings in Informatics (LIPIcs), A. V. Papadopoulos, Ed., vol. 262. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, pp. 16:1–16:21.

[2] ——, "Open Problem Resolved: The "Two" in Existing Multiprocessor PI-Blocking Bounds Is Fundamental," in *ECRTS*, ser. Leibniz International Proceedings in Informatics (LIPIcs), R. Pellizzoni, Ed., vol. 298. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, pp. 11:1–11:21.

[3] J. Anderson and A. Srinivasan, "Mixed pfair/erfair scheduling of asynchronous periodic tasks," in *Proceedings 13th Euromicro Conference on Real-Time Systems*, 2001, pp. 76–85.

[4] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson, "A flexible real-time locking protocol for multiprocessors," in *RTCSA*, 2007, pp. 47–56.

[5] B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, University of North Carolina at Chapel Hill, 2011.

[6] B. B. Brandenburg, "The fmlp+: An asymptotically optimal real-time locking protocol for suspension-aware analysis," in *ECRTS*. Los Alamitos, CA, USA: IEEE Computer Society, Jul 2014, pp. 61–71.

[7] B. B. Brandenburg and J. H. Anderson, "Optimality results for multiprocessor real-time locking," in *RTSS*, 2010, pp. 49–60.

[8] ——, "Real-time resource-sharing under clustered scheduling: mutex, reader-writer, and k-exclusion locks," in *EMSOFT*, 2011, pp. 69–78.

[9] ——, "The OMLP family of optimal multiprocessor real-time locking protocols," *Des. Autom. Embedded Syst.*, vol. 17, no. 2, p. 277–342, Jun. 2013.

[10] B. B. Brandenburg, "A fully preemptive multiprocessor semaphore protocol for latency-sensitive real-time applications," in *ECRTS*, 2013, pp. 292–302.

[11] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson, "LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers," in *RTSS*, 2006, pp. 111–126.

[12] M. Cirinei and T. Baker, "EDZL scheduling analysis," in *ECRTS*, 08 2007, pp. 9–18.

[13] J. Corbet, "An eevdf cpu scheduler for linux." [Online]. Available: https://lwn.net/Articles/925371/

[14] P. Emberson, R. Stafford, and R. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, July 2010, pp. 6–11.

[15] J. P. Erickson and J. H. Anderson, "Reducing tardiness under global scheduling by splitting jobs," in *ECRTS*, 2013, pp. 14–24.

[16] P. Holman and J. Anderson, "Locking under Pfair scheduling," *ACM Transactions on Computer Systems*, vol. 24, no. 2, pp. 140–174, 2006.

[17] J. Lee and I. Shin, "EDZL schedulability analysis in real-time multicore scheduling," vol. 39, no. 7, p. 910–916, Jul. 2013.

[18] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[19] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C. Plaxton, "A proportional share resource allocation algorithm for real-time, time-shared systems," in *RTSS*, 1996, pp. 288–299.

[20] Z. Tong, S. Ahmed, and J. Anderson, "Holistically budgeting processing graphs," in *RTSS*, 2023, pp. 27–39.

[21] Z. Tong, S. Ali, and J. Anderson, "Asymptotically optimal multiprocessor real-time locking for non-JLFP scheduling," *full version with appendix*. [Online]. Available: https://jamesanderson.web.unc.edu/papers/

[22] M. Yang, A. Wieder, and B. B. Brandenburg, "Global real-time semaphore protocols: A survey, unified analysis, and comparison," in *RTSS*, 2015, pp. 1–12.

APPENDIX

Here, we prove that the lower bound on maximum pi-blocking achieved by the considered arbitrary suspension-based mutex locking protocol under clustered non-JLFP scheduling is $\Omega(m + m(H_n - H_m))$. To achieve the maximum pi-blocking lower bound we partition Γ evenly among the $\frac{m}{c}$ clusters, each with c processors. To avoid notational complexity, we assume that the n tasks in Γ can be evenly partitioned among the $\frac{m}{c}$ clusters. Thus, the set of tasks in each cluster u, denoted by Γ_u, contains $n_u = \frac{cn}{m}$ tasks.

Congruency between clustered and global scheduling. Under clustered scheduling, the schedule of Γ_u in the u^{th} cluster (which we henceforth refer to as cluster u) is identical to the schedule of Γ_u under a global scheduler on a system containing c processors. As such, by considering only jobs and requests from the cluster u, we can borrow much of the analysis in Sec. III. We therefore define Def. 6 and Rules L3 and L4 in the context of cluster u.

Definition 17. *Let $\mathcal{R}^{i,u}$ be the i^{th} request to become satisfied on cluster u among $\{\mathcal{R}_x \mid \tau_x \in \Gamma_u\}$ by the considered locking protocol. We denote the time $\mathcal{R}^{i,u}$ becomes satisfied as $t_{i,u}$.*

Recall from Def. 6 that t_i is the time that the i^{th} request becomes satisfied among $\{R_x \mid t_x \in \Gamma\}$. Using this and Def. 17, we give the following definition that aids in the description of our modified Rules L3 and L4.

Definition 18. *Let each $t'_{0,u} = 0$ and let $t'_{i,u}$ where $i \geq 1$ equal $min(t_{m(i-1)/c+u}, t_{i,u})$.*

Priority change rules. Using Def. 18, we present our priority change rules. The reason behind Def. 18 and Rules L3' and L4' will become apparent later.

> **L3'** In each time interval $[t'_{i,u}, t'_{i+1,u})$ where $i \in [n_u - c, n_u)$, $J(\mathcal{R}^{j,u})$'s priority, where $j \in (i, n_u]$, becomes the $(j-i)^{th}$ highest among all pending jobs.
>
> **L4'** For any integers $i \in [0, n_u - c)$ and $k \in [0, n_u - i)$, in each time interval $I_{i,u,k} =$
>
> $$\left[t_{i,u} + \frac{k(t'_{i+1,u} - t'_{i,u})}{n_u - i}, t_{i,u} + \frac{(k+1)(t'_{i+1,u} - t'_{i,u})}{n_u - i}\right),$$
>
> the h^{th} highest priority pending job where $h \in [1, c]$ is job $J(\mathcal{R}^{j,u})$, where $j = i + 1 + (h - 1 + kc \bmod n_u - i)$.

We can see that Rules L3' and L4' come about by replacing n with n_u, m with c, $\mathcal{R}^j$ with $\mathcal{R}^{j,u}$, and t_i with $t'_{i,u}$. Thus, through a similar proof process, we can obtain the following.

Lemma 15. *$J(\mathcal{R}^{n_u,u})$ in cluster u is pi-blocked for*

$$\sum_{i=0}^{n_u-c-1} \frac{c(t'_{i+1,u} - t'_{i,u})}{n_u - i} + \sum_{i=n_u-c}^{n_u-1} t'_{i+1,u} - t'_{i,u} \tag{9}$$

in the time interval $[t'_{0,u}, t'_{n_u,u})$.

To determine the minimum time in which $J(\mathcal{R}^{n_u,u})$ is pi-blocked, we proceed to minimize (9). We begin letting $\mathcal{L}$ denote the minimum value of each $t_{i+1} - t_i$. Since at most one request can be satisfied at any time under a mutual exclusion locking protocol, Rules L2 and Def. 18 ensure the following.

Property 5. *Under mutual exclusion locking, $\mathcal{L} \geq 1$.*

Additionally, the definition of $\mathcal{L}$ also implies the following.

Property 6. $t_{mi/c+u} - t_{m(i-1)/c+u} = \frac{m\mathcal{L}}{c}$.

Ideal vs non-ideal scenarios. We denote the scenario when each $t'_{i,u} = t_{m(i-1)/c+u}$ where $i \geq 1$ and each $t'_{0,u} = 0$ as the *ideal scenario*. Contrastingly, we call the scenario when some $t'_{i,u} \neq t_{m(i-1)/c+u}$ or some $t'_{0,u} \neq 0$ as the *non-ideal scenario*. In fact, we can show that in the ideal scenario, $J(\mathcal{R}^{n_{m/c},m/c})$ in cluster m/c experiences less pi-blocking compared to in non-ideal scenarios. We demonstrate this by examining the difference between each $t'_{i+1,u} - t'_{i,u}$ term when under the ideal and non-ideal scenarios. This difference, denoted as $\delta_{i,u}$, is formally defined as follows.

Definition 19. *For each value of $i \geq 0$ and u, let $\delta_{i,u} = (t'_{i+1,u} - t'_{i,u}) - (t_{mi/c+u} - t_{m(i-1)/c+u})$. Additionally, each $\delta_{0,u} = t'_{1,u} - t_u$.*

By substitution into (9), we can see that $J(\mathcal{R}^{n_u,u})$ in cluster u is pi-blocked equal to the following in $[t_{0,u}, t_{n_u,u})$.

$$\frac{ct_u + \delta_{0,u}}{n_u} + \sum_{i=1}^{n_u-c-1} \frac{c(t_{mi/c+u} - t_{m(i-1)/c+u} + \delta_{i,u})}{n_u - i} + \sum_{i=n_u-c}^{n_u-1} t_{mi/c+u} - t_{m(i-1)/c+u} + \delta_{i,u} \tag{10}$$

Using (10), we can demonstrate that the ideal scenario does indeed result in the minimum pi-blocking for $J(\mathcal{R}^{n_{m/c},m/c})$.

Lemma 16. *Consider Γ under clustered scheduling where job priorities satisfy Rule L3' and L4', $J(\mathcal{R}^{n_{m/c},m/c})$ is pi-blocked for at least*

$$\sum_{i=0}^{n_{m/c}-c-1} \frac{m\mathcal{L}}{n_{m/c} - i} + \sum_{i=n_{m/c}-c}^{n_{m/c}-1} \frac{m\mathcal{L}}{c} \tag{11}$$

in the interval $[t_{0,m/c}, t_{n_{m/c},m/c})$

Proof. WLOG, we assume that the last request to be satisfied is on cluster m/c. This implies that $\mathcal{R}^{n_{m/c},m/c}$ becomes satisfied at t_n. We can write $n = m/c \cdot cn/m$, thus since $n_{m/c} = cn/m$, we have $t'_{n_{m/c},m/c} = t_n = t_{m/c(n_{m/c})}$.

First, let x be an index where $t'_{x,m/c} \neq t_{m(x-1)/c+m/c}$. By Def. 18, $t_{x,m/c} < t_{mx/c}$, implying that $t_{x,m/c} = t_{mx/c-k}$ for some positive integer k. Since $t'_{n_{m/c},m/c} = t_{m/c(n_{m/c})}$, we have $t'_{n_{m/c},m/c} - t'_{x,m/c} = (m/c(n_{m/c} - x) + k)\mathcal{L}$. Thus, by Def. 19 and Prop. 6, we have $\sum_{i=x}^{n_{m/c}-1} \delta_{i,m/c} = k\mathcal{L}$. On the other hand, since $t'_{0,m/c} = 0$ by Def. 18, we have $t'_{x,m/c} - t'_{0,m/c} = (mx/c - k)\mathcal{L}$. Thus, by Def. 19 and Prop. 6, we

have $\sum_{i=0}^{x-1} \delta_{i,m/c} = -k\mathcal{L}$. We now consider the following two cases: $x \leq n_{m/c} - c - 1$, and $x \geq n_{m/c} - c$.

Case 1. $x \leq n_{m/c} - c - 1$. In this case, we can see that

$$\sum_{i=0}^{x-1} \frac{c \cdot \delta_{i,m/c}}{n_{m/c} - i} \geq \frac{-cx\mathcal{L}}{n_{m/c} - x + 1}$$

$$\sum_{i=x}^{n_{m/c}-1} \frac{c \cdot \delta_{i,m/c}}{n_{m/c} - i} \geq \frac{cx\mathcal{L}}{n_{m/c} - x}$$

both hold. This implies that the sum of the two above summations is at least zero.

Case 2. $x \geq n_{m/c} - c$. In this case, we can see that

$$\sum_{i=0}^{x-1} \frac{c \cdot \delta_{i,m/c}}{n_{m/c} - i} \geq -cx\mathcal{L}$$

$$\sum_{i=x}^{n_{m/c}-1} \frac{c \cdot \delta_{i,m/c}}{n_{m/c} - i} \geq cx\mathcal{L}$$

both hold. Thus, the sum of the above two summations is also at least zero. Hence, in both cases, $\sum_{i=0}^{n_{m/c}-1} \frac{c \cdot \delta_{i,m/c}}{n_{m/c} - i} \geq 0$. By examining (10), we see that this summation accounts for all the delta terms. Thus, since this summation is at least zero, $J(\mathcal{R}^{n_{m/c},m/c})$ is pi-blocked for at least

$$\frac{ct_{m/c}}{n_{m/c}} + \sum_{i=1}^{n_{m/c}-c-1} \frac{c(t_{mi/c+m/c} - t_{m(i-1)/c+m/c})}{n_{m/c} - i} + \sum_{i=n_{m/c}-c}^{n_{m/c}-1} t_{mi/c+m/c} - t_{m(i-1)/c+m/c} \tag{12}$$

By Props. 5 and 6, (12) simplifies to (11). □

Using this lemma, we can prove the following lower bound on pi-blocking.

Theorem 4. *There exists a task set where a job using any mutual exclusion locking protocol and under clustered non-JLFP scheduling can incur at least* $\Omega(m + m(H_n - H_m))$ *pi-blocking.*

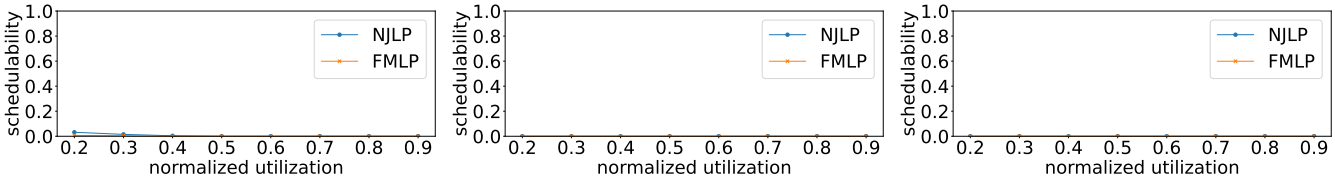
Proof. Consider the pathological task set Γ under clustered non-JLFP scheduling. From Def. 1, job priorities can change according to Rules L3' and L4'. Now, consider the pi-blocking of $J(\mathcal{R}^{n_{m/c},m/c})$ under a mutual exclusion locking protocol when job priorities change according to Rules L3' to L4'. Due to Lem. 15 and Prop. 5, $J(\mathcal{R}^{n_{m/c},m/c})$ is pi-blocked for at least the value of (11) in the time interval $[t_{0,m/c}, t_{n_{m/c},m/c})$. By multiplying both the numerator and denominator of each term by $\frac{m}{c}$, and substituting n_u by $\frac{cn}{m}$, we have

$$\sum_{i=0}^{cn/m-c} \sum_{j=1}^{m/c} \frac{m\mathcal{L}}{\frac{m}{c}(\frac{cn}{m} - i)} + \sum_{i=cn/m-c+1}^{cn/m-1} \sum_{j=1}^{m/c} \frac{m\mathcal{L}}{m}$$

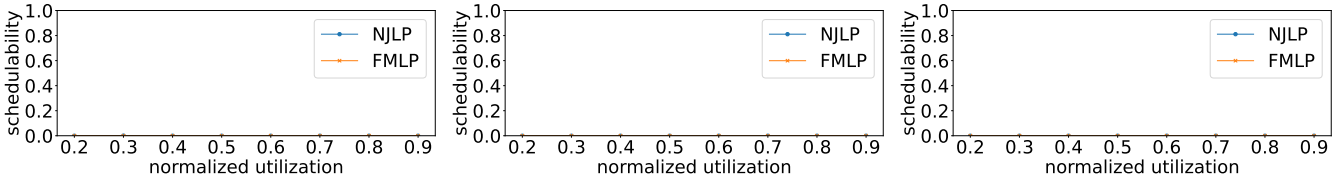
$$\geq \sum_{i=0}^{cn/m-c} \sum_{j=1}^{m/c} \frac{m\mathcal{L}}{n - \frac{m(i-1)}{c} - j} + \sum_{i=n-m+1}^{n-m/c} \mathcal{L}$$

$$= \sum_{i=1}^{n-m} \frac{m\mathcal{L}}{n + m/c - i} + \sum_{i=n-m+1}^{n-m/c} \mathcal{L} \tag{13}$$

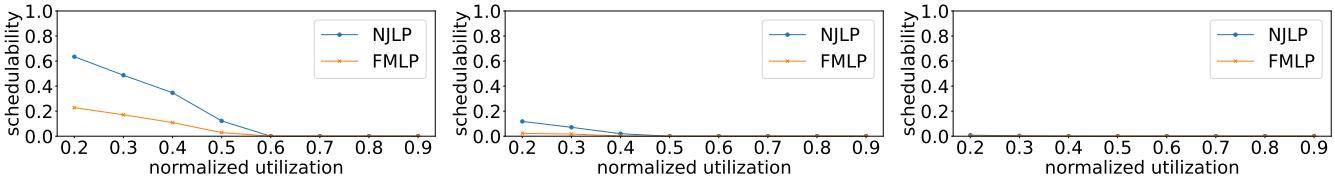
By Def. 5, the first term of (13) is equal to $m\mathcal{L}(H_{n+m/c-1} - H_m)$. Additionally, the second term of (13) is equal to $(m - m/c)\mathcal{L}$. Therefore, $J(\mathcal{R}^n)$ is pi-blocked for at least $((m - m/c) + m(H_{n+m/c-1} - H_m))\mathcal{L}$ in $[t_{0,m/c}, t_{n_{m/c},m/c})$. This implies that $J(\mathcal{R}^n)$ is pi-blocked for at least $((m - m/c) + m(H_{n+m/c-1} - H_m))\mathcal{L}$, which implies $\Omega(m + m(H_n - H_m))$. lower-bounds the pi-blocking of a job. □



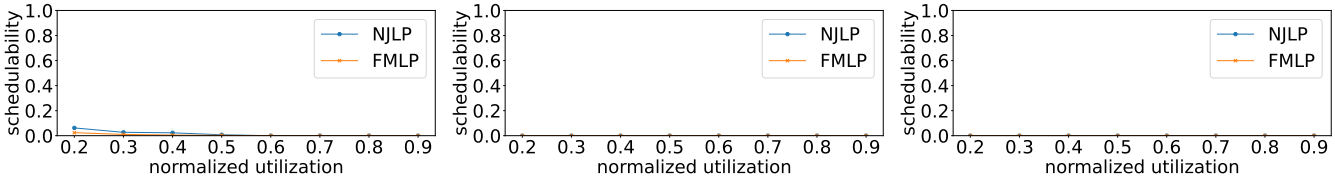
(a) Large task set, short periods, short requests, 1 resource. (b) Large task set, short periods, short requests, 2 resources. (c) Large task set, short periods, short requests, 3 resources.



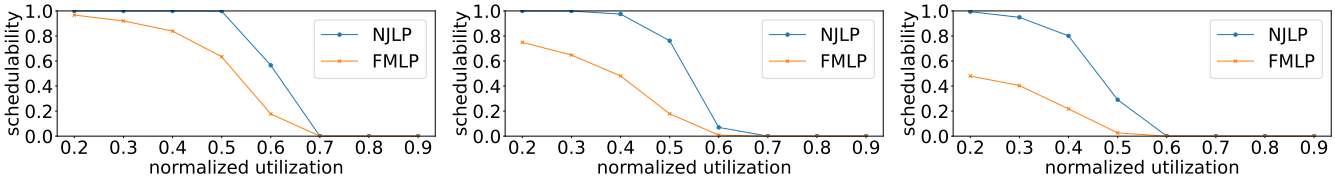
(d) Large task set, short periods, long requests, 1 resource. (e) Large task set, short periods, long requests, 2 resources. (f) Large task set, short periods, long requests, 3 resources.



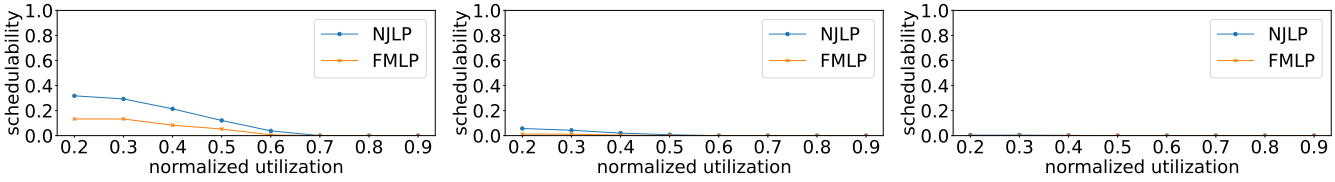
(g) Large task set, medium periods, short requests, 1 resource. (h) Large task set, medium periods, short requests, 2 resources. (i) Large task set, medium periods, short requests, 3 resources.



(j) Large task set, medium periods, long requests, 1 resource. (k) Large task set, medium periods, long requests, 2 resources. (l) Large task set, medium periods, long requests, 3 resources.

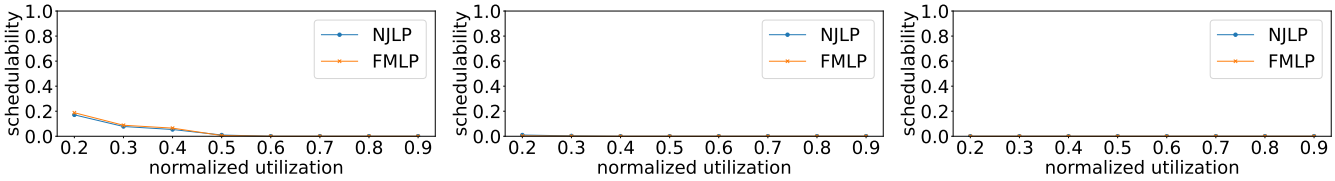


(m) Large task set, long periods, short requests, 1 resource. (n) Large task set, long periods, short requests, 2 resources. (o) Large task set, long periods, short requests, 3 resources.



(p) Large task set, long periods, long requests, 1 resource. (q) Large task set, long periods, long requests, 2 resources. (r) Large task set, long periods, long requests, 3 resources.

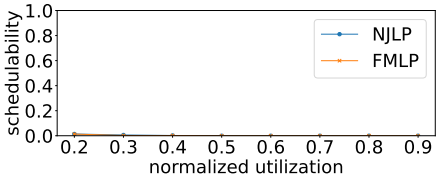
Fig. 11: Results of our overhead-aware schedulability study for large task sets.



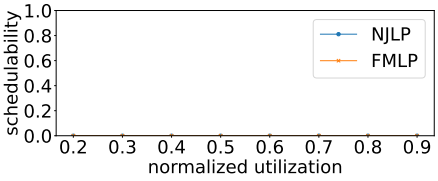
(a) Medium task set, short periods, short requests, 1 resource.

(b) Medium task set, short periods, short requests, 2 resources.

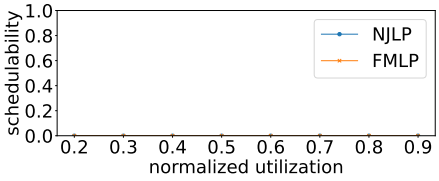
(c) Medium task set, short periods, short requests, 3 resources.



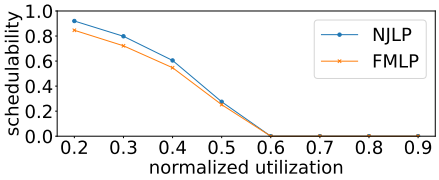
(d) Medium task set, short periods, long requests, 1 resource.



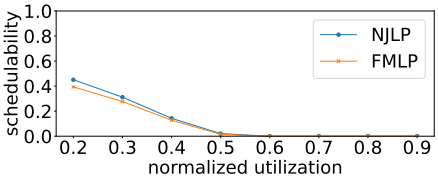
(e) Medium task set, short periods, long requests, 2 resources.



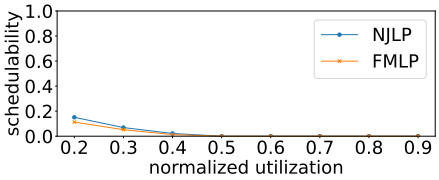
(f) Medium task set, short periods, long requests, 3 resources.



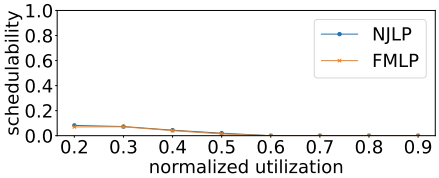
(g) Medium task set, medium periods, short requests, 1 resource.



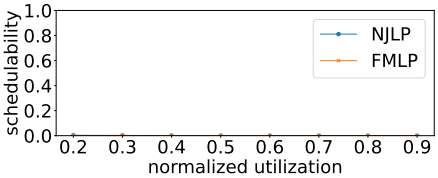
(h) Medium task set, medium periods, short requests, 2 resources.



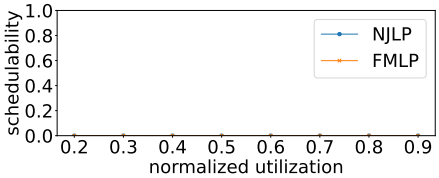
(i) Medium task set, medium periods, short requests, 3 resources.



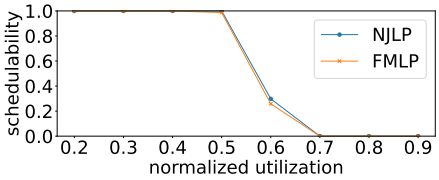
(j) Medium task set, medium periods, long requests, 1 resource.



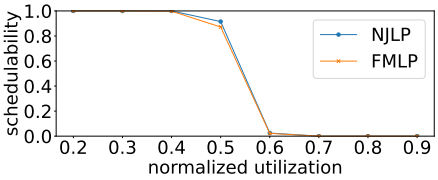
(k) Medium task set, medium periods, long requests, 2 resources.



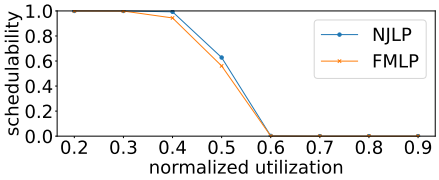
(l) Medium task set, medium periods, long requests, 3 resources.



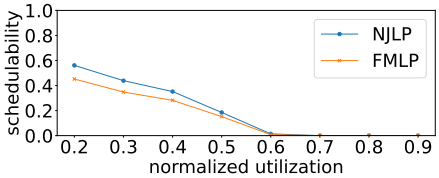
(m) Medium task set, long periods, short requests, 1 resource.



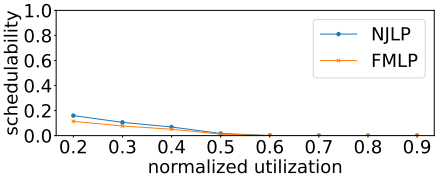
(n) Medium task set, long periods, short requests, 2 resources.



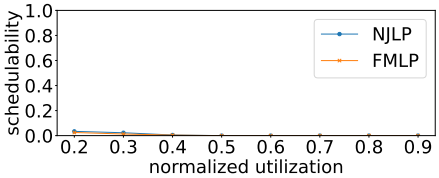
(o) Medium task set, long periods, short requests, 3 resources.



(p) Medium task set, long periods, long requests, 1 resource.

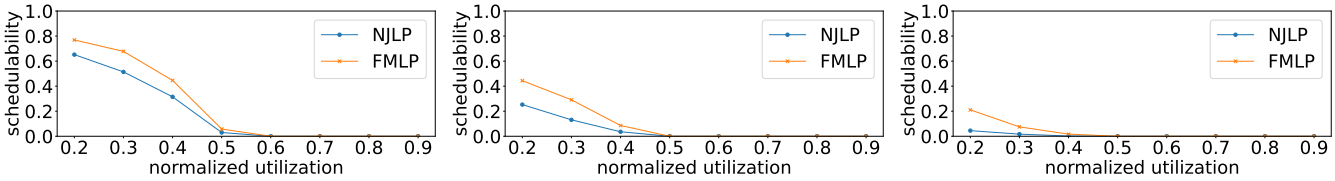


(q) Medium task set, long periods, long requests, 2 resources.

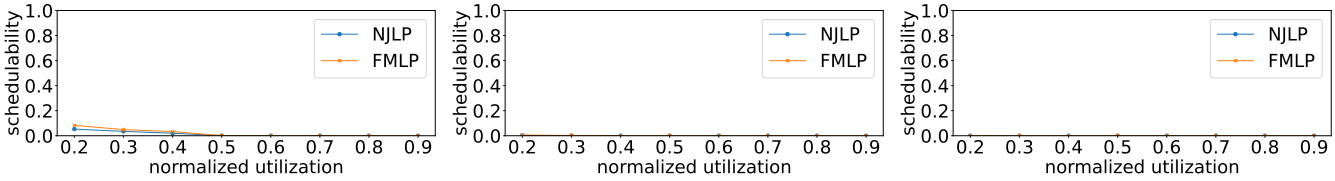


(r) Medium task set, long periods, long requests, 3 resources.

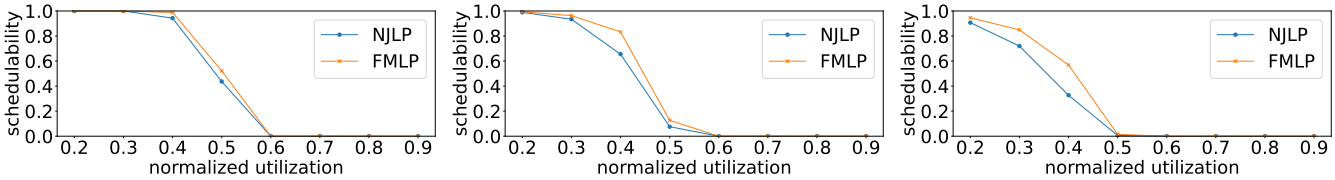
Fig. 12: Results of our overhead-aware schedulability study for medium task sets.



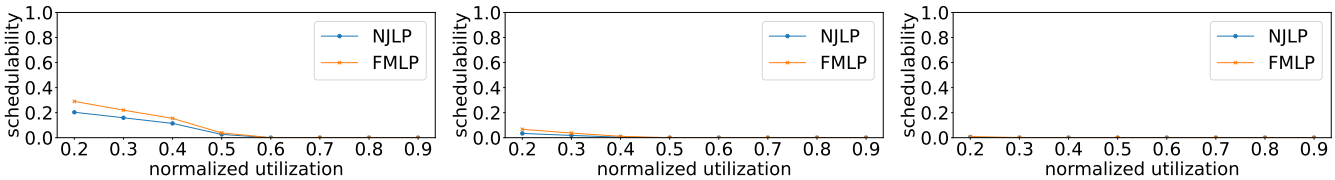
(a) Small task set, short periods, short requests, 1 resource. (b) Small task set, short periods, short requests, 2 resources. (c) Small task set, short periods, short requests, 3 resources.



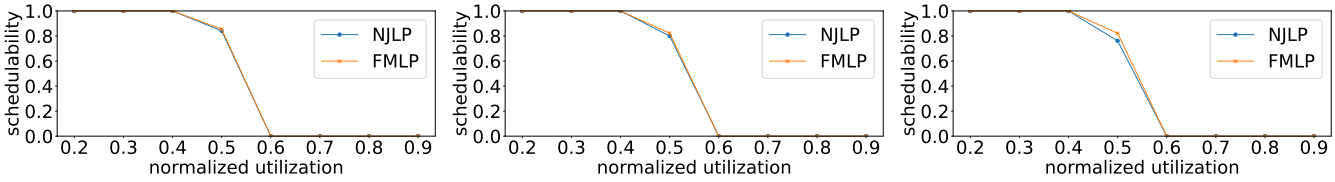
(d) Small task set, short periods, long requests, 1 resource. (e) Small task set, short periods, long requests, 2 resources. (f) Small task set, short periods, long requests, 3 resources.



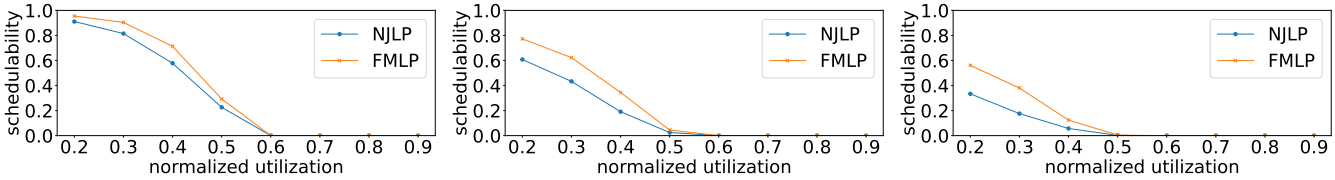
(g) Small task set, medium periods, short requests, 1 resource. (h) Small task set, medium periods, short requests, 2 resources. (i) Small task set, medium periods, short requests, 3 resources.



(j) Small task set, medium periods, long requests, 1 resource. (k) Small task set, medium periods, long requests, 2 resources. (l) Small task set, medium periods, long requests, 3 resources.



(m) Small task set, long periods, short requests, 1 resource. (n) Small task set, long periods, short requests, 2 resources. (o) Small task set, long periods, short requests, 3 resources.



(p) Small task set, long periods, long requests, 1 resource. (q) Small task set, long periods, long requests, 2 resources. (r) Small task set, long periods, long requests, 3 resources.

Fig. 13: Results of our overhead-aware schedulability study for small task sets.