Optimality Results for Multiprocessor Real-Time Locking*

Björn B. Brandenburg and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

When locking protocols are used in real-time systems, bounds on blocking times are required when ensuring timing constraints. While the term "blocking" is well-understood in the context of uniprocessor real-time systems, the same is not true in the multiprocessor case. In this paper, two definitions of blocking are presented that are applicable to suspension-based multiprocessor locking protocols. The need for two definitions arises because of differences in how suspensions are handled in existing schedulability analysis. For each definition, locking protocols are presented that have asymptotically optimal blocking behavior. In particular, protocols are presented for any job-level static-priority global or partitioned scheduling algorithm.

1 Introduction

The recent shift by major chip manufacturers to multicore technologies has led to renewed interest in infrastructure and analysis techniques for supporting multiprocessor real-time applications. In order to support such applications, multiprocessor real-time scheduling algorithms and resource-sharing protocols are required that, when used together, enable a task system's timing constraints to be ensured. Ensuring such constraints usually requires restricting the supported task system in some way. For example, it may be necessary to restrict per-task utilizations or overall system utilization. With regard to lock-based resource sharing, such restrictions arise because of processor capacity that is lost when tasks *block* on one another as they wait to acquire shared resources. A good locking protocol should minimize such loss.

In the uniprocessor case, good locking protocols are well known. Indeed, uniprocessor protocols exist that ensure that each job (instance) of a task blocks for the duration of at most one (outermost) critical section, which is obviously asymptotically optimal [1, 21, 23]. In the multiprocessor case, however, the situation is much more murky, despite the considerable body of work on multiprocessor real-time locking protocols (which we review below). In fact, to the best of our knowledge, general, *precise* definitions of what actually constitutes "blocking" in this case do not even exist. Rather, existing protocols have been analyzed by providing upper bounds on lock-acquisition delays that would be sufficient under any reasonable definition of blocking. It goes without saying that without a precise definition of blocking, we clearly have no understanding of what constitutes *optimal* blocking behavior in multiprocessor systems.

Motivated by this, we discuss in this paper how to precisely define blocking in the multiprocessor case and present multiprocessor real-time locking protocols that have asymptotically optimal blocking behavior, *i.e.*, protocols under which the amount of time lost to blocking (by *any* task set) is bounded within a constant factor of the loss shown to be unavoidable in the worst case (for some task sets). We specifically focus on implicit-deadline sporadic task systems that are scheduled by job-level static-priority schedulers [11]. We consider three such schedulers in detail: global and partitioned *earliest-deadline-first* scheduling (G-EDF and P-EDF, resp.), in which jobs are prioritized in earliest-deadline-first order, and partitioned *static-priority* scheduling (P-SP), in which each task is assigned a fixed priority. Regarding resource sharing, our focus is locking protocols in which tasks wait by suspending. We assume that lock accesses are not nested, or equivalently, nested accesses are realized by using group locks [8].

Prior work. Due to space constraints, our discussion of prior work is not exhaustive, but rather focuses on those prior efforts that are of most relevance to the results we present. Rajkumar *et al.* [20, 21, 22] were the first to propose locking protocols for real-time multiprocessor systems. They presented two suspension-based protocols for P-SP-scheduled systems, the *multiprocessor priority-ceiling protocol* (MPCP) [17, 20] and the *distributed priority-ceiling protocol* (DPCP) [22]. In later work on P-EDF-scheduled systems, Chen and Tripathi [12] presented two protocols that only apply to periodic (and not sporadic) tasks, Lopez *et al.* [19] presented a partitioning heuristic that transforms global resources (*i.e.*, resources that can be accessed from multiple processors) into local resources, and Gai *et al.* [15] proposed a protocol in which blocking for global resources is realized via spinning (*i.e.*, busy-waiting) rather than suspending. More recently, Block *et al.* [8] presented the *flexible multiprocessor locking protocol* (FMLP), which can be used under G-EDF, P-EDF, and P-SP [9]. The FMLP categorizes critical sections as either "short" or "long": blocking is realized by spinning (suspension) for short (long) critical sections. Finally, Easwaran and Andersson [14] recently pre-

*Work supported by AT&T and IBM Corps.; NSF grants CNS 0834270 and CNS 0834132; ARO grant W911NF-09-1-0535; and AFOSR grant FA 9550-09-1-0549.

sented the suspension-based *parallel priority-ceiling protocol* (PPCP) and an analysis of the *priority inheritance protocol* (PIP) for globally-scheduled static-priority systems.

In all of the just-cited papers, the focus is on developing locking protocols for which blocking times can be sufficiently bounded. Issues of optimality are not considered. In many cases, the bounds that are derived are quite pessimistic. Pessimism will inevitably arise, for example, if a locking protocol is used that makes it difficult to disambiguate true blocking from ordinary demand, *i.e.*, demand for processor time that must be accounted for assuming no locks exist. In such cases, lock-acquisition delays may get doubly charged as both blocking and ordinary demand. Of course, disambiguating blocking from demand requires formal definitions.

Contributions. As in the uniprocessor case, we argue that, with respect to multiprocessor locking protocols, true blocking that must be accounted for arises when *priority inversions* occur. Accordingly, we use the term *pi-blocking*. Because the definition of pi-blocking is rooted in the notion of a "priority inversion," a formal definition of the former requires a formal definition of the latter. While the notion of a priority inversion is straightforward to define in the uniprocessor case, we argue that an appropriate definition in the multiprocessor case hinges on whether schedulability analysis is *suspension-oblivious* or *suspension-aware*. In brief (see Sec. 2), suspensions are modeled as ordinary computation under suspension-oblivious analysis, but as true suspensions under suspension-aware analysis.

Our complexity bounds apply to a system of n implicit-deadline sporadic tasks scheduled by a job-level, static-priority scheduler on m processors, where the number of critical sections per job and the length of each critical section are taken to be constant. In the suspension-oblivious case, we present an $\Omega(m)$ lower bound on per-job worst-case blocking that applies to both partitioned and global schedulers (see Sec. 3.1). We also present global (Sec. 3.2) and partitioned (Sec. 3.3) variants of a new optimal locking protocol, the $O(m)$ *locking protocol* (OMLP) (Secs. 3.2 and 3.3), for which per-job blocking times are $O(m)$. These protocols have better blocking times than prior algorithms under suspension-oblivious analysis, so for them, we provide more exact (not just asymptotic) blocking analysis.

In the suspension-aware case, we show that $O(m)$ blocking complexity is not possible by establishing an $\Omega(n)$ lower bound on blocking (Sec. 4.1). We show that the FMLP is optimal in the global case (Sec. 4.2), and present a simple, optimal FIFO algorithm that, in the partitioned case, has $O(n)$ blocking complexity under suspension-aware analysis (Sec. 4.3). The simplicity of this algorithm derives from the fact that it greatly limits parallelism in accessing resources. One may question whether an asymptotically better approach might be possible by ordering requests on a static-priority or EDF basis. We show that this is not possible by establishing an $\Omega(mn)$ lower bound that is applicable to any such approach.

We formalize our system model and summarize relevant background next.

2 Background and Definitions

We consider the problem [11, 18] of scheduling a set of n implicit-deadline[1] sporadic tasks $\tau = \{T_1, \ldots, T_n\}$ on m processors; we let $T_i(e_i, p_i)$ denote a task with a *worst-case per-job execution time* e_i and a *minimum job separation* p_i. $J_{i,j}$ denotes the j^{th} job ($j \geq 1$) of T_i. $J_{i,j}$ is *pending* from its arrival (or release) time $a_{i,j} \geq 0$ until it finishes execution at time $f_{i,j}$. If $j > 1$, then $a_{i,j} \geq a_{i,j-1} + p_i$. $J_{i,j}$'s *response time* is given by $f_{i,j} - a_{i,j}$. We omit the job index j if it is irrelevant and let J_i denote an arbitrary job.

For a given scheduling algorithm A, task T_i's *worst-case response time* r_i is the maximum response time of any job of T_i in any schedule of τ produced by A. A task set is *schedulable* under A if $r_i \leq p_i$ for each T_i, *i.e.*, if every job completes by its implicit deadline [18].

A pending job is in one of two states: a *ready* job is available for execution, whereas a *suspended* job cannot be scheduled. A job *resumes* when its state changes from suspended to ready. We assume that pending jobs are ready unless suspended by a locking protocol.

Resources. The system contains q *shared resources* $\ell_1, \ldots, \ell_q$ (such as shared data objects and I/O devices) besides the m processors. When a job J_i requires a resource ℓ_k, it *issues* a *request* $\mathcal{R}$ for ℓ_k. $\mathcal{R}$ is *satisfied* as soon as J_i *holds* ℓ_k, and *completes* when J_i *releases* ℓ_k. The *request length*, denoted $\|\mathcal{R}\|$, is the time that J_i must execute[2] before it releases ℓ_k. We let $N_{i,k}$ denote the maximum number of times that any J_i requests ℓ_k, and let $L_{i,k}$ denote the maximum length of such a request, where $L_{i,k} = 0$ if $N_{i,k} = 0$.

A resource can be held by at most one job at any time. Thus, a *locking protocol* must be employed to order conflicting requests. A job J_i that issues a request $\mathcal{R}$ incurs *acquisition delay* and cannot proceed with its computation while it waits for $\mathcal{R}$ to be satisfied. There are two principle mechanisms to realize waiting: a job can either *busy-wait* (or *spin*) in a tight loop, thereby wasting processor time, or it can relinquish the processor and *suspend* until its request is satisfied.

A resource ℓ_k is *local* to a processor P if all jobs requesting ℓ_k execute on P, and *global* otherwise. Local resources can be optimally managed with uniprocessor protocols [1, 21]; the focus of this paper is global resources.

We assume non-nested resource requests, *i.e.*, jobs request at most one resource at any time. We note, however, that nesting can be handled with group locks as in the FMLP [8], albeit at the expense of reduced parallelism.

[1]The presented results do not depend on the choice of deadline constraint. Implicit deadlines were chosen to avoid irrelevant detail.

[2]We assume that J_i must be scheduled to complete its request. This is required for shared data objects, but may be pessimistic for I/O devices. The latter can be accounted for at the expense of more verbose notation.

Scheduling. All schedulers considered in this paper are assumed to be work-conserving *job-level static-priority* (*JLSP*) *schedulers* [11]. We consider three such schedulers in detail: global and partitioned *earliest-deadline-first* scheduling (G-EDF and P-EDF, resp.), in which jobs are prioritized in order of increasing deadline (with ties broken in favor of lower-index tasks), and partitioned *static-priority* scheduling (P-SP), in which each task is assigned a fixed priority. We assume that P-SP-scheduled tasks are indexed in order of decreasing priority. Tasks (and their jobs) are statically assigned to processors under partitioning; in this case, we let P_i, $1 \leq P_i \leq m$, denote T_i's assigned processor, and let $part(x) \triangleq \{T_i \mid P_i = x\}$ denote the set of tasks assigned to processor x. Under global scheduling, jobs are scheduled from a single ready queue and may migrate [11].

Locking protocols may temporarily raise a job's effective priority. Under *priority inheritance* [21, 23], the effective priority of a job J_i holding a resource ℓ_k is the maximum of J_i's priority and the priorities of all jobs waiting for ℓ_k. Alternatively, under *priority boosting* [9, 17, 20, 21, 22], a resource-holding job's priority is unconditionally elevated above the highest-possible base (*i.e.*, non-boosted) priority to expedite the request completion. Non-preemptive sections can be understood as a form of priority boosting.

Blocking. For historical reasons, "blocking" is an overloaded term. In non-real-time settings, jobs waiting for a shared resource are commonly said to be "blocked." In the context of uniprocessor real-time resource sharing, "blocking" has a more specific meaning: a waiting job is not blocked whenever the currently-scheduled job is of higher priority [1, 21, 23]. This notion of blocking arises because acquisition delay can increase response times and must be accounted for when determining whether a task set is schedulable. Since acquisition delay that overlaps with higher-priority work does not affect response times, it is not counted as "blocking" even though the job is "blocked on" a resource. In this interpretation, a job incurs "blocking" only during times of *priority inversion* [21, 23], *i.e.*, if a low-priority job is scheduled while a higher-priority job is pending.

Further, in the context of schedulability analysis, "blocking" can also refer to other delays unrelated to resource sharing that cause response-time increases. For example, this includes *deferral blocking* [24], which arises under static-priority scheduling due to suspensions. Deferral blocking does not necessarily coincide with priority inversion.

In this paper, we consider the definition specific to resource sharing, which we denote as *priority inversion blocking* (*pi-blocking*) to avoid ambiguity. To reiterate, pi-blocking occurs whenever a job's completion is delayed and this delay cannot be attributed to higher-priority demand (formalized below). We let b_i denote a bound on the total pi-blocking incurred by any J_i.

Before we continue, we need to clarify the concept of a "priority inversion on a multiprocessor," which is complicated by two issues. First, on a uniprocessor, pi-blocking occurs when a low-priority job is scheduled in place of a higher-priority job. This intuitive definition does not generalize to multiprocessors: as some processors may idle while a job is waiting, pi-blocking may be incurred even when no lower-priority job is scheduled.

Second, multiprocessor schedulability analysis has not yet matured to the point that suspensions can be analyzed under all schedulers. In particular, none of the seven major G-EDF hard real-time schedulability tests [2, 3, 4, 5, 6, 7, 16] inherently accounts for suspensions. Such analysis is *suspension-oblivious* (*s-oblivious*): jobs may suspend, but each e_i must be inflated by b_i prior to applying the test to account for all additional delays. This approach is safe—converting execution time to idle time does not increase response times—but pessimistic, as even suspended jobs are (implicitly) considered to prevent lower-priority jobs from being scheduled. In contrast, *suspension-aware* (*s-aware*) schedulability analysis that explicitly accounts for b_i is available for global static-priority scheduling, P-EDF, and P-SP (*e.g.*, see [14, 17, 21]). Notably, suspended jobs are *not* considered to occupy a processor under s-aware analysis.

Consequently, priority inversion is defined differently under s-aware and s-oblivious analysis: since suspended jobs are counted as demand under s-oblivious analysis, the mere *existence* of m pending higher-priority jobs rules out a priority inversion, whereas only *ready* higher-priority jobs can nullify a priority inversion under s-aware analysis.

Def. 1. Under global **s-oblivious** schedulability analysis, a job J_i incurs *s-oblivious pi-blocking* at time t if J_i is pending but not scheduled and fewer than m higher-priority jobs are **pending**.

Def. 2. Under global **s-aware** schedulability analysis, a job J_i incurs *s-aware pi-blocking* at time t if J_i is pending but not scheduled and fewer than m higher-priority jobs are **ready**.[3]

In both cases, "higher-priority" is interpreted with respect to base priorities. The difference between s-oblivious and s-aware pi-blocking is illustrated in Fig. 1. Notice that Def. 1 is weaker than Def. 2. Thus, lower bounds on s-oblivious pi-blocking apply to s-aware pi-blocking as well, and the converse is true for upper bounds.

In the case of partitioning, definitions similar to Defs. 1 and 2 apply on a per-processor basis, *i.e.*, only local higher-priority jobs are considered and $m = 1$.

Blocking complexity. We study two characteristic complexity metrics that reflect overall pi-blocking: *maximum pi-blocking*, $\max_{T_i \in \tau}\{b_i\}$, which reflects per-task bounds that are required for schedulability analysis, and *total pi-blocking*, $\sum_{i=1}^{n} b_i$, which yields average pi-blocking and thus provides context for the maximum, *i.e.*, it indicates

[3] Easwaran and Andersson [14] provide a definition of "job blocking" that conceptually resembles our notion of s-aware pi-blocking. However, their definition specifically applies to global static-priority scheduling and does not encompass all of the effects that we consider to be "blocking" (*e.g.*, such as priority boosting).

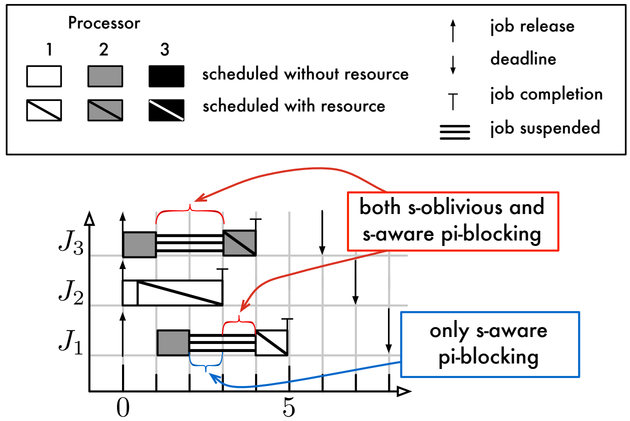

Figure 1: Example of s-oblivious and s-aware pi-blocking of three jobs J_1, J_2, and J_3, sharing one resource on two G-EDF-scheduled processors. J_3 suffers acquisition delay during $[1, 3)$, and since no higher-priority jobs exist it is pi-blocked under either definition. J_1, suspended during $[2, 4)$, suffers pi-blocking under either definition during $[3, 4)$ since it is among the m highest-priority pending jobs, but only s-aware pi-blocking during $[2, 3)$ since J_3 is pending but not ready then. (The notation in this figure is used in subsequent figures as well.)

whether the maximum pi-blocking bound is "typical."

Assumptions. Concrete bounds on pi-blocking must necessarily depend on each $L_{i,k}$—long requests will cause long priority inversions under any protocol. Similarly, bounds under any protocol become increasingly pessimistic as the total number of requests per job grows. Thus, when deriving asymptotic bounds, we consider, for each T_i, $\sum_{1 \le k \le q} N_{i,k}$ and each $L_{i,k}$ to be constants and assume $n \ge m$. All other parameters are considered variable (or dependent on m and n). In particular, we do not impose constraints on the ratio $\max\{p_i\}/\min\{p_i\}$ or the number of tasks sharing each ℓ_k.

3 S-Oblivious Pi-Blocking

We first consider s-oblivious pi-blocking. We begin by establishing lower bounds on maximum and total pi-blocking, and then present an optimal locking protocol for both global (Sec. 3.2) and partitioned scheduling (Sec. 3.3).

3.1 Lower Bound

$\Omega(m)$ pi-blocking is unavoidable in some cases. Consider the following pathological high-contention task set.

Def. 3. Let $\tau^{seq}(n)$ denote a task set of n identical tasks that share one resource ℓ_1 such that $e_i = 1$, $p_i = 2n$, $N_{i,1} = 1$, and $L_{i,1} = 1$ for each T_i, where $n \ge m \ge 2$.

Lemma 1. *There exists an arrival sequence for $\tau^{seq}(n)$ such that, under s-oblivious analysis, $\max_{T_i \in \tau}\{b_i\} = \Omega(m)$ and $\sum_{i=1}^{n} b_i = \Omega(nm)$ under* any *locking protocol and JLSP scheduler.*

Proof. Without loss of generality, assume that n is an integer multiple of m. Consider the schedule resulting from the following periodic arrival sequence: each $J_{i,j}$ is released at time $a_{i,j} = (\lceil i/m \rceil - 1) \cdot m + (j-1) \cdot p_i$, and issues one request $\mathcal{R}_{i,j}$, where $\|\mathcal{R}_{i,j}\| = 1$, *i.e.*, releases occur in groups of m jobs and each job requires ℓ_1 for its entire computation. The resulting G-EDF schedule is illustrated in Fig. 2.

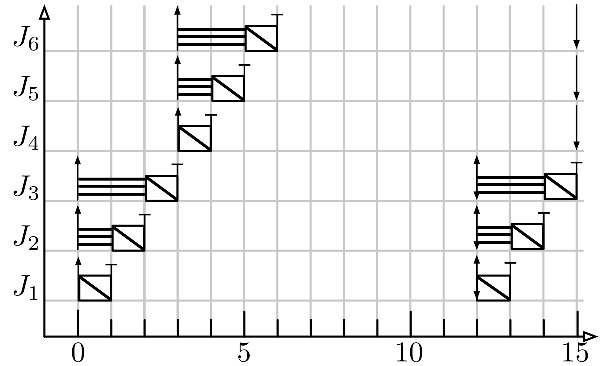

Figure 2: G-EDF schedule of $\tau^{seq}(n)$ for $n = 6$ and $m = 3$, and thus $g \in \{0, 1\}$. The first group of jobs ($J_{1,1}, J_{2,1}, J_{3,1}$) is released at time 0; the second group ($J_{4,1}, J_{5,1}, J_{6,1}$) is released at time 3. Each group incurs $0 + 1 + 2 = \sum_{i=0}^{m-1} i$ total pi-blocking.

There are n/m groups of m tasks each that release jobs simultaneously. Each group of jobs of $T_{g \cdot m+1}, \ldots, T_{g \cdot m+m}$, where $g \in \{0, \ldots, n/m - 1\}$, issues m concurrent requests for ℓ_1. Since ℓ_1 cannot be shared, any locking protocol must impart some order, and thus there exists a job in each group that incurs d time units of pi-blocking for each $d \in \{0, \ldots, m-1\}$. Hence, for each g, $\sum_{i=g \cdot m+1}^{g \cdot m+m} b_i \ge \sum_{i=0}^{m-1} i = \Omega(m^2)$, and thus, across all groups, $\sum_{i=1}^{n} b_i = \sum_{g=0}^{(n/m-1)} \sum_{i=g \cdot m+1}^{g \cdot m+m} b_i \ge n/m \cdot \Omega(m^2) = \Omega(nm)$, which implies $\max_{T_i \in \tau}\{b_i\} = \Omega(m)$.

By construction, the schedule does not depend on G-EDF scheduling since no more than m jobs are pending at any time, and thus applies to other global JLSP schedulers as well. The lower bound applies equally to partitioned JLSP schedulers since $\tau^{seq}(n)$ can be trivially partitioned such that each processor serves at least $\lfloor n/m \rfloor$ and no more than $\lceil n/m \rceil$ tasks. □

Prior work shows this bound to be tight for spin-based protocols—if jobs busy-wait non-preemptively in FIFO order, then they must wait for at most $m - 1$ earlier requests [8, 13]. However, prior work has not yielded an $O(m)$ suspension-based protocol.

3.2 Optimal Locking under Global Scheduling

As mentioned in the introduction, Block *et al.*'s FMLP [8] is the only prior locking protocol for G-EDF that allows waiting jobs to suspend. The FMLP's primary design goal is *simplicity*, in both implementation and analysis. Accordingly, conflicting requests for both short and long resources are satisfied in FIFO order. As pointed out above, this is optimal for busy-waiting. In contrast, the FMLP analysis for long resources is asymptotically worse—jobs can incur $O(n)$ pi-blocking when waiting for a long resource [8], and

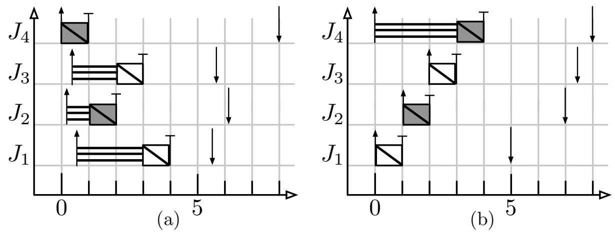

Figure 3: Two examples of $n = 4$ tasks sharing one resource ℓ_1 on $m = 2$ G-EDF-scheduled processors. Each job requires ℓ_1 for the entirety of its computation. **(a)** If conflicting requests are satisfied in FIFO order, then the job with the earliest deadline (J_1) may incur $\Omega(n)$ pi-blocking if its request is issued just after all other requests. **(b)** If conflicting requests are satisfied in order of job priority, then a job's request may be deferred repeatedly even though it is among the m highest-priority jobs.

schedules in which a job does indeed incur $\Theta(n)$ s-oblivious pi-blocking are readily created, as shown in Fig. 3(a).

It is tempting to view this as an indictment of FIFO ordering, as one might reasonably expect a real-time locking protocol to reflect job priorities. However, ordering requests by job priority, as done in [14], does not improve the bound: since a low-priority job can be starved by later-issued higher-priority requests, it is easy to construct an arrival sequence in which a job incurs $\Omega(n)$ s-oblivious pi-blocking, as seen in Fig. 3(b). Thus, ordering *all* requests by job priority is, at least asymptotically speaking, not preferable to the much-simpler FIFO queuing.

Fortunately, by combining FIFO and priority ordering, it is possible to realize $O(m)$ pi-blocking, as shown next.

3.2.1 The Global OMLP

The $O(m)$ *locking protocol* (*OMLP*) is a suspension-based resource sharing protocol in which jobs incur at most $O(m)$ s-oblivious pi-blocking. In the global OMLP, each resource is protected by two locks: a priority-based m-exclusion lock[4] that limits access to a regular FIFO mutex lock, which in turn serializes access to the resource.

Structure. For each resource ℓ_k, there are two job queues: FQ_k, a FIFO queue of length at most m, and PQ_k, a priority queue (ordered by job priority) that is only used if more than m jobs are contending for ℓ_k. The job at the head of FQ_k (if any) holds ℓ_k.

Rules. Let $queued_k(t)$ denote the number of jobs queued in both FQ_k and PQ_k at time t. Requests are ordered according to the following rules.

G1 A job J_i that issues a request $\mathcal{R}$ for ℓ_k at time t is appended to FQ_k if $queued_k(t) < m$; otherwise, if $queued_k(t) \geq m$, it is added to PQ_k. $\mathcal{R}$ is satisfied when J_i becomes the head of FQ_k.

[4]An m-exclusion lock can be held concurrently by up to m jobs.

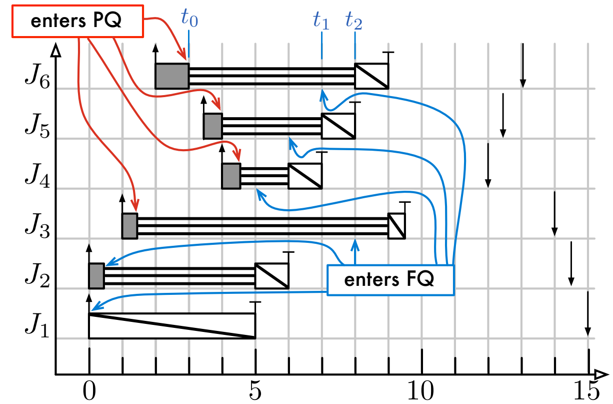

Figure 4: Example showing the global OMLP under G-EDF for six tasks sharing one resource on $m = 2$ processors. J_6 issues a request at $t_0 = 3$, enters FQ_1 at $t_1 = 7$, and holds ℓ_1 at $t_2 = 8$. Note that J_1 and J_2 enter FQ_1 immediately for lack of contention, and thus J_2's request precedes J_4's request in spite of J_4 having an earlier deadline. In contrast, J_4 and J_5 arrive and enqueue after J_6, but enter FQ_1 before J_6 due to their earlier deadlines and Rule G3. Similarly, J_6 acquires ℓ_k before J_3, despite J_3's earlier request.

G2 All queued jobs are suspended, with the exception of the job at the head of FQ_k, which is ready and inherits the priority of the highest-priority job in FQ_k and PQ_k.

G3 When J_i releases ℓ_k, it is dequeued from FQ_k and the new head of FQ_k (if any) is resumed. Also, if PQ_k is non-empty, then the highest-priority job in PQ_k is moved to FQ_k.

The key insight is the use of an m-exclusion lock to safely defer requests of lower-priority jobs without allowing a pi-blocked job to starve. This can be observed in the example depicted in Fig. 4. At time 1.5, $m = 2$ jobs hold the m-exclusion lock, *i.e.*, have entered FQ_1, and thus J_3 must enter PQ_1. Hence it is safely deferred when ℓ_1 is later requested by higher-priority jobs (J_4, J_5, J_6). At the same time, J_2, which incurs pi-blocking until J_6's arrival at time 2, precedes the later-issued requests since it already held the m-exclusion lock—this avoids starvation in scenarios such as the one depicted in Fig. 3(b).

3.2.2 Global OMLP Schedulability Analysis

We first derive a bound on the number of requests that cause J_i to be pi-blocked. We then show how such bounds affect scheduling analysis by considering G-EDF.

In the following, let t_0 denote the time at which J_i issues $\mathcal{R}$, t_1 denote the time at which J_i enters FQ_k, and t_2 denote the time at which $\mathcal{R}$ is satisfied (see Fig. 4). Further, let $entered(t)$, $t_0 \leq t < t_1$, denote the number of jobs that have been moved from PQ_k to FQ_k during $[t_0, t]$ due to Rule G3, *i.e.*, that preceded J_i in entering FQ_k. For example, for J_6 in Fig. 4, $entered(3.5) = 0$ and $entered(6) = 2$.

Lemma 2. *For each point in time $t \in [t_0, t_1)$, if J_i is pi-blocked at time t, then $entered(t) < m$.*

Proof. By Rule G3, because J_i has not yet entered FQ_k at time t, there must be m pending jobs queued in FQ_k. Due to FIFO ordering, if $entered(t) \geq m$, then each job queued in FQ_k at time t must have been enqueued in FQ_k during $[t_0, t]$. By Rule G3, this implies that each job in FQ_k must have a priority that exceeds J_i's priority. By the definition of s-oblivious pi-blocking (Def. 1), the presence of m higher-priority pending jobs implies that J_i is not pi-blocked. □

Lemma 3. *During $[t_0, t_2]$, J_i incurs pi-blocking for the combined duration of at most $2 \cdot (m-1)$ requests.*

Proof. Due to the bounded length of FQ_k, at most $m-1$ requests complete in $[t_1, t_2]$ before a given request is satisfied. By Lemma 2 and Rule G3, at most $m-1$ requests complete before J_1 is no longer pi-blocked in $[t_0, t_1)$. □

Combining Lemma 3 with the maximum request length for each ℓ_k yields the following bound.

Lemma 4. *J_i is pi-blocked for at most*

$$b_i \triangleq \sum_{k=1}^{q} N_{i,k} \cdot 2 \cdot (m-1) \cdot \max_{1 \leq i \leq n} \{L_{i,k}\}.$$

Proof. By Lemma 3, J_i is pi-blocked for the duration of at most $2 \cdot (m-1)$ requests each time it requests a resource ℓ_k. Due to priority inheritance, the resource-holding job has an effective priority among the m highest priorities whenever J_i is pi-blocked; requests are thus guaranteed to progress towards completion when J_i is pi-blocked. As J_i requests ℓ_k at most $N_{i,k}$ times, it suffices to consider the longest request for ℓ_k $N_{i,k} \cdot 2 \cdot (m-1)$ times. The sum of the per-resource bounds yields b_i. □

Theorem 1. *S-oblivious pi-blocking under the global OMLP is asymptotically optimal.*

Proof. Follows from Lemmas 1, 3, and 4. □

Practically speaking, the bound given in Lemma 4 can be pessimistic since it does not take the actual "demand" for shared resources into account, *i.e.*, this bound cannot reflect low-contention scenarios in which each ℓ_k is requested by only few tasks. For example, consider a message buffer that is shared between only two tasks and suppose $m = 100$: assuming that every request is interfered with by 198 requests is clearly needlessly pessimistic. We provide a less pessimistic bound that reflects individual request frequencies and lengths in Appendix A.

Recall that b_i was derived assuming that suspended higher-priority jobs are accounted for as demand. Thus, each per-job execution time must be inflated by b_i before applying existing G-EDF schedulability tests that assume tasks to be independent.

Theorem 2. *A task set τ is schedulable under G-EDF and the OMLP if τ' is deemed schedulable by an s-oblivious G-EDF schedulability test for independent tasks [2, 3, 4, 5, 6, 7, 16], where $\tau' = \{T_i'(e_i + b_i, p_i) \mid T_i \in \tau\}$.*

Note that the derivation of b_i itself does not depend on G-EDF; the OMLP can thus also be applied to other global JLSP schedulers.

3.3 Optimal Locking under Partitioned Scheduling

Additional challenges arise under partitioning since priority inheritance across partitions is, from an analytical point of view, ineffective. Under global scheduling (and on uniprocessors), priority inheritance ensures that the resource-holding job has sufficient priority to be scheduled whenever a waiting job is pi-blocked. In contrast, the highest *local* priority may be lower than the priority of any *remote* job under partitioning and thus progress cannot be guaranteed.

In prior work, three partitioned, suspension-based real-time locking protocols have been proposed: the DPCP [21, 22] and MPCP [17, 20, 21] for P-SP scheduling, and the FMLP for both P-EDF [8] and P-SP [9] scheduling. These protocols share two characteristics: they all employ priority boosting instead of (or in addition to) priority inheritance, and they use global, per-resource wait queues, in which jobs are ordered either by priority (DPCP and MPCP) or in FIFO order (FMLP). Interestingly, either design choice can result in schedules with $\Omega(n)$ pi-blocking for some jobs. This is avoided by the partitioned OMLP.

3.3.1 The Partitioned OMLP

Since comparisons of local and remote priorities cannot be used to bound pi-blocking under partitioning, the partitioned OMLP uses a *token abstraction* to limit global contention. A "contention token" is a virtual, local resource that a job must hold before it may request a global resource. There is only a single contention token per processor, *i.e.*, the same token is used for all global resources. This serves to limit the number of jobs that can cause pi-blocking due to priority boosting.

Structure. The *contention token* CT_P local to processor P, $P \in \{1, \ldots, m\}$, is a binary semaphore with an associated priority queue PQ_P (ordered by job priority). There is one global FIFO queue FQ_k of length at most m for each resource ℓ_k. The job at the head of FQ_k holds ℓ_k.

Rules. Let J_i denote a job on processor P_i that issues a request $\mathcal{R}$ for a global resource ℓ_k at time t.

- **P1** If CT_{P_i} is not held by any (local) job at time t, then J_i acquires CT_{P_i} and proceeds with Rule P3. Otherwise, J_i is suspended and enqueued in PQ_{P_i}.
- **P2** If J_i was suspended due to Rule P1, then it resumes and acquires CT_{P_i} at the earliest point in time such that both **(a)** J_i is the highest-priority pending job assigned to P_i and **(b)** CT_{P_i} is not being held.
- **P3** Once J_i holds CT_{P_i}, it is added to FQ_k. J_i is suspended unless FQ_k was empty before adding J_i.
- **P4** J_i's effective priority is boosted while holding CT_{P_i}, *i.e.*, J_i is scheduled unless suspended.

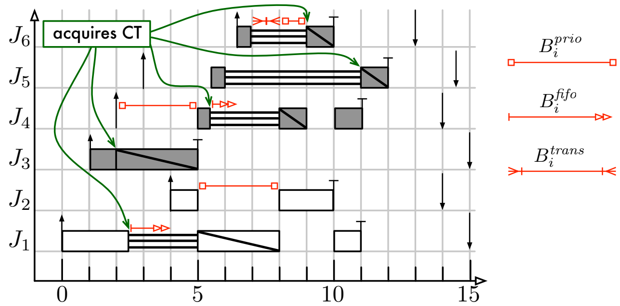

Figure 5: Illustration of s-oblivious pi-blocking under P-EDF and the partitioned OMLP for six tasks sharing one resource on $m = 2$ processors. J_1 incurs direct pi-blocking (B_i^{fifo}) while waiting for J_3 to release ℓ_1 until the higher-priority job J_2 is released at time 4. J_2 is preempted and incurs pi-blocking (B_i^{prio}) when J_1 is priority-boosted during $[5, 8)$. J_4 incurs pi-blocking immediately on release at time 2 because J_3 is priority-boosted (B_i^{prio}), and again during $[5.5, 6.5]$ while waiting for J_1 to release ℓ_1 (B_i^{fifo}). J_4 is no longer pi-blocked when the higher-priority J_6 is released at time 6.5. J_6 incurs pi-blocking during $[7, 9)$ while it waits for J_6 to release CT_2. The first half of the interval is accounted for by B_6^{trans} since J_6 is transitively pi-blocked by a remote request while J_4 is waiting itself. This changes at time 8 when J_4 is priority-boosted to complete its request, which is accounted for by B_6^{prio}. Note that J_5 does not resume until time 11 when it becomes the highest-priority pending job despite CT_2 becoming available at time 10.

P5 When J_i releases ℓ_k, it is removed from FQ_k, releases CT_{P_i}, and the new head of FQ_k (if any) is resumed.

An example schedule is depicted in Fig. 5. Under the partitioned OMLP, jobs that do not share resources themselves may still incur pi-blocking due to priority boosting (*e.g.*, this happens to J_2 at time 5 in Fig. 5). This is not the case under the global OMLP, which highlights the advantage of using priority inheritance if possible.

Note that the set of all m contention tokens implements an m-exclusion algorithm—thus, at most m jobs may contend for global resources at any time. This property is essential to obtaining the following $O(m)$ bound on pi-blocking.

3.3.2 Partitioned OMLP Schedulability Analysis

Pi-blocking arises in three ways under the partitioned OMLP, as illustrated in Fig. 5. J_i may incur pi-blocking

1. due to a **local** request if it is preempted by a priority-boosted job (Rule P4)—this delay is denoted B_i^{prio};
2. *directly* due to **remote** requests while waiting in a FIFO queue (Rule P3)—this delay is denoted B_i^{fifo}; and
3. *transitively* due to **remote** requests while waiting for CT_{P_i} (Rules P1 and P2) if the CT_{P_i}-holding job is suspended itself (Rule P3)—this delay is denoted B_i^{trans}.

We bound each of these sources individually and begin with interference due to lower-priority jobs.

Lemma 5. *No job local to P_i with priority lower than J_i's priority acquires CT_{P_i} while J_i is pending.*

Proof. Suppose a lower-priority job J_x acquires CT_{P_i} at time t while J_i is pending, *i.e.*, $t \in [a_i, f_i)$. If J_x was suspended by Rule P1, then, by Rule P2(a), J_x cannot resume and acquire CT_{P_i} at time t. Thus, to issue a request at time t, J_x must be ready and scheduled, which implies that J_i is suspended. If J_i is suspended due to Rule P4, then it holds CT_{P_i} itself. If J_i is suspended due to Rule P1, then CT_{P_i} is not available at time t. Thus, in either case J_x cannot acquire CT_{P_i} at time $t \in [a_i, f_i)$. □

Lemma 6. *J_i is pi-blocked due to local, priority-boosted, lower-priority jobs (Rule P4) for at most*

$$B_i^{prio} \triangleq \max\{L_{x,k} \mid T_x \in part(P_i) \wedge 1 \leq k \leq q\}.$$

Proof. Let J_x denote a lower-priority job that is priority-boosted while J_i is pending. By Rule P4, J_x must hold CT_{P_i}. By Lemma 5, J_x must have acquired CT_{P_i} before J_i's release. At most one such J_x can exist. J_x releases CT_{P_i} after one request (Rule P5). Thus, J_i is blocked for the length of at most one local request. □

Next, we bound pi-blocking due to Rule P3, which only affects jobs that issue requests.

Lemma 7. *While holding CT_{P_i}, J_i incurs pi-blocking for at most*

$$B_i^{fifo} \triangleq \sum_{k=1}^{q} N_{i,k} \cdot (m-1) \cdot \max_{1 \leq x \leq n}\{L_{x,k}\}.$$

Proof. Due to Rule P1, at most one job on every remote processor can globally contend at any time. Thus, due to the FIFO ordering of each FQ_k, at most $(m-1)$ requests precede J_i's request each time that J_i requires ℓ_k. Priority boosting ensures that the resource-holding job is always scheduled (Rule P4), thus progress is ensured. Since FIFO queues are not shared among resources, the sum of the individual per-resource bounds yields B_i^{fifo}. □

Finally, we need to bound pi-blocking due to Rule P1.

Lemma 8. *While waiting for CT_{P_i}, J_i incurs at most*

$$B_i^{trans} \triangleq (m-1) \cdot \max_{1 \leq k \leq q} \max_{1 \leq x \leq n}\{L_{x,k}\}$$

pi-blocking due to requests issued by remote jobs.

Proof. J_i must wait for CT_{P_i} if it is held by either a higher-priority or a lower-priority local job J_x. By the definition of s-oblivious pi-blocking, J_i is only pi-blocked in the latter case. By Lemma 5, this can occur at most once. While J_i waits for J_x to release CT_{P_i}, it is transitively pi-blocked by at most $(m-1)$ remote requests since at most $m-1$ jobs can precede J_x in the FIFO queue. Priority boosting ensures the progress of resource-holding jobs. □

Note that B_i^{prio} already accounts for the execution of the lower-priority job's request, and that B_i^{trans} thus only accounts for pi-blocking that J_i incurs while the CT_{P_i}-holding job is suspended (*e.g.*, see J_6 in Fig. 5). Further, note that B_i^{trans} and B_i^{fifo} only apply to tasks that share resources, *i.e.*, if $\sum_{k=1}^{q} N_{i,k} > 0$.

Lemma 9. *J_i incurs pi-blocking for at most $b_i \triangleq B_i^{prio}$ if $\sum_{k=1}^{q} N_{i,k} = 0$, i.e., if T_i does not access resources, and $b_i \triangleq B_i^{prio} + B_i^{fifo} + B_i^{trans}$ otherwise.*

Proof. Follows from the preceding discussion. □

Note that the above bound is again very coarse-grained and thus likely pessimistic. As in the global case, we provide a less pessimistic bound that reflects individual request frequencies and lengths in Appendix A. However, Lemma 9 suffices to establish optimality.

Theorem 3. *S-oblivious pi-blocking under the partitioned OMLP is asymptotically optimal.*

Proof. Recall from Sec. 2 that $\sum_k N_{i,k}$ and each $L_{i,k}$ are assumed constant. It follows that $B_i^{prio} = O(1)$, $B_i^{fifo} = O(m)$, and $B_i^{trans} = O(m)$, and hence $\max_{T_i \in \tau}\{b_i\} = O(m)$ and $\sum_{i=1}^{n} b_i = O(nm)$. □

Schedulability under P-EDF can be established with the classic s-oblivious EDF utilization bound [18].

Theorem 4. *A task set τ is schedulable under P-EDF and the OMLP if, for each processor o, $1 \leq o \leq m$, $\sum_{T_i \in part(o)} \frac{e_i + b_i}{p_i} \leq 1$.*

As in the global case, the derivation of b_i does not inherently depend on EDF scheduling, and can be applied to other JLSP schedulers by substituting an appropriate s-oblivious schedulability test.

4 S-Aware Pi-Blocking

One can easily construct schedules with later-arriving, higher-priority jobs similar to Fig. 3(b) that demonstrate that the OMLP does not ensure $O(m)$ *s-aware* pi-blocking. Naturally, the question arises: can the OMLP be "tweaked" to achieve this bound? This is, in fact, impossible.

4.1 Lower Bound

The following lemma shows that maximum s-aware pi-blocking of $\Omega(n)$ is fundamental.

Lemma 10. *There exists an arrival sequence for $\tau^{seq}(n)$ (see Def. 3) such that, under s-aware analysis, $\max_{T_i \in \tau}\{b_i\} = \Omega(n)$ and $\sum_{i=1}^{n} b_i = \Omega(n^2)$, under any locking protocol and JLSP scheduler.*

Proof. Without loss of generality, assume that n is an integer multiple of m. We first consider the partitioned case and assume that $P_i = \lceil i/m \rceil$, *i.e.*, n/m tasks are assigned to each processor.

Consider the schedule S^{seq} resulting from a synchronous, periodic arrival sequence: each $J_{i,j}$ is released at $a_{i,j} = (j-1) \cdot p_i$, and issues one request $\mathcal{R}_{i,j}$, where $\|\mathcal{R}_{i,j}\| = 1$. S^{seq} is illustrated in Fig. 6(a) assuming P-EDF scheduling.

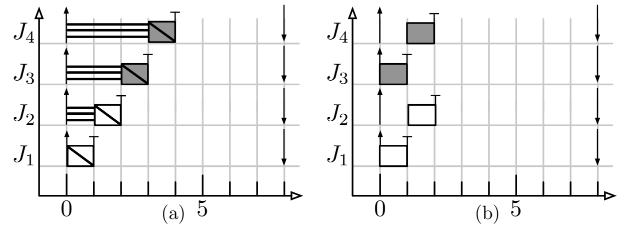

Figure 6: Illustration of **(a)** S^{seq} and **(b)** S^{par} for $n = 4$ and $m = 2$ under P-EDF. Note that $b_i \geq r_i^{seq} - r_i^{par}$. For example, J_4 incurs pi-blocking during $[0, 2)$ in S^{seq}; consequently $b_4 \geq r_4^{seq} - r_4^{par} = 4 - 2 = 2$. Similarly, $b_3 \geq r_3^{seq} - r_3^{par} = 3 - 1 = 2$.

Note that linear suspension times are immediately apparent. However, to bound b_i under *any* JSLP scheduler, we need to take into account the times in which a suspended job is not pi-blocked because a higher-priority job executes.

Towards this aim, consider the schedule S^{par} resulting from the same arrival sequence assuming that jobs are independent, *i.e.*, each J_i executes for e_i without requesting ℓ_1. S^{par} is illustrated in Fig. 6(b).

Under S^{seq}, because jobs are serialized by ℓ_1, only one job completes every time unit until no jobs are pending; thus, $\sum_{i=1}^{n} r_i^{seq} = \sum_{i=1}^{n} i$ irrespective of how requests are ordered or jobs prioritized.

Under S^{par}, because jobs are independent and the scheduler is, by assumption, work-conserving, m jobs complete concurrently every time unit until no jobs are pending; thus, under any job prioritization, $\sum_{i=1}^{n} r_i^{par} = \sum_{i=1}^{n} \lceil i/m \rceil$.

By construction, no job is pi-blocked in S^{par}. In contrast, jobs incur pi-blocking in S^{seq} under the *same JLSP scheduler*, *i.e.*, jobs are prioritized consistently in S^{par} and S^{seq}. Thus, the observed response time increase of every job reflects the amount of pi-blocking incurred in S^{seq}. Therefore, for each T_i, $b_i \geq r_{i,1}^{seq} - r_{i,1}^{par}$, and thus $\sum_{i=1}^{n} b_i \geq \sum_{i=1}^{n} r_{i,1}^{seq} - \sum_{i=1}^{n} r_{i,1}^{par} = \sum_{i=1}^{n} i - \sum_{i=1}^{n} \lceil \frac{i}{m} \rceil \geq \sum_{i=1}^{n} i - \frac{1}{m} \sum_{i=1}^{n} i - \sum_{i=1}^{n} 1 = \left(1 - \frac{1}{m}\right)(n+1)\frac{n}{2} - n = \Omega(n^2)$. This implies $\max_{T_i \in \tau}\{b_i\} = \Omega(n)$.

Since at most one job is scheduled in S^{seq} at any time, pi-blocking does not decrease under global scheduling. □

We next show this bound to be tight under both global and partitioned scheduling.

4.2 Optimal Locking under Global Scheduling

The suspension-based "long" FMLP for G-EDF [8] uses per-resource FIFO queues with priority inheritance, *i.e.*, there is a FIFO queue FQ_k for each resource ℓ_k, J_i is appended to FQ_k when requesting ℓ_k, and the job at the head of FQ_k holds ℓ_k and inherits the priority from any job blocked on ℓ_k.[5] This, in fact, ensures asymptotically optimal s-aware pi-blocking.

Theorem 5. *S-aware pi-blocking under the global, suspension-based FMLP [8] is asymptotically optimal.*

[5]This is a somewhat simplified—but faithful—description of the suspension-based "long" FMLP; see Block *et al.* [8] for details. Our discussion does not apply to the spin-based "short" FMLP.

Proof. We derive a simple bound on s-aware pi-blocking under the FMLP.

Each time that J_i requests a resource ℓ_k, it is enqueued in FQ_k. Thus, it must wait for the completion of at most $n-1$ requests. Due to priority inheritance, the job at the head of FQ_k is guaranteed to be scheduled whenever J_i is pi-blocked. Thus, J_i incurs pi-blocking for at most

$$b_i \triangleq \sum_{k=1}^{q} N_{i,k} \cdot (n-1) \max_{1 \leq x \leq n} \{L_{x,k}\}$$

across all requests. Since $\max_{1 \leq x \leq n}\{L_{x,k}\}$ and $\sum_{k=1}^{q} N_{i,k}$ are considered constant (see Sec. 2), this implies that $\max_{T_i \in \tau}\{b_i\} = O(n)$ and thus $\sum_{i=1}^{n} b_i = O(n^2)$. □

This implies that the bound established in Lemma 10 is tight under global scheduling. Further note that, even though the FMLP was originally proposed for G-EDF, the above analysis does not depend on G-EDF and can be applied to other global JLSP schedulers as well.

4.3 Optimal Locking under Partitioned Scheduling

As discussed in Sec. 3.3, the lack of (in an analytical sense) effective priority inheritance under partitioning complicates matters. In particular, in the case of the *partitioned* FMLP, it is easy to show a (likely pessimistic) bound of $O(n^2)$ maximum pi-blocking, but the FMLP's reliance on priority boosting makes it challenging to derive a tighter bound.

However, it turns out that a much simpler protocol suffices to establish tightness of the $\Omega(n)$ lower bound. Consider the following *simple, partitioned FIFO locking protocol* (SPFP): there is only one global FIFO queue FQ_G that is used to serialize requests to *all* requests, and the job at the head of the queue is priority-boosted.

Theorem 6. *S-aware pi-blocking under the SPFP is asymptotically optimal.*

Proof. Analogously to Thm. 5: each request is preceded by at most $n-1$ requests, and the job at the head of FQ_G is guaranteed to be scheduled since it is the only priority-boosted job. Thus, J_i incurs pi-blocking under the SPFP for at most

$$b_i \triangleq \max_{1 \leq x \leq n} \{L_{x,k} \mid 1 \leq k \leq q\} \cdot (n-1) \cdot \sum_{k=1}^{q} N_{i,k}.$$

The theorem follows. □

The "trick" behind the SPFP is to avoid pessimism that arises when multiple jobs sharing a processor are concurrently priority-boosted. Obviously, serializing *all* requests is of only limited practical value. However, the asymptotic optimality of the SPFP does establish that $\Omega(n)$ s-aware pi-blocking is a tight lower bound in the general case.

4.4 Lower Bound on EDF and Static-Priority Queuing

Intuitively, one might reasonably expect queuing disciplines that order lock requests on a static-priority or EDF basis to

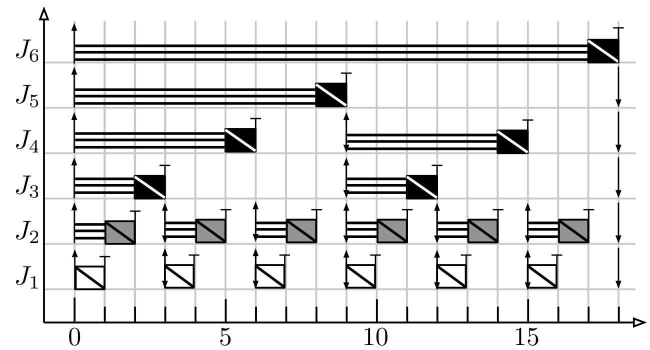

Figure 7: Illustration of $\tau^{prio}(n)$ for $n = 6$ and $m = 3$. The depicted schedule arises under any partitioned JLSP scheduler if requests are ordered either by deadline or by static priority (with task indexed in order of decreasing priority). Also, an equivalent schedule arises under global scheduling since only one job is scheduled at any time. J_6 incurs pi-blocking for $(m-1) \cdot n = 12$ time units under partitioning and for $mn - 1 = 17$ time units under global scheduling (throughout $[0, 17)$ only one job is scheduled).

cause no more blocking than simple FIFO queuing. However, asymptotically speaking, this is not the case.

Consider the following task set.

Def. 4. Let $\tau^{prio}(n)$, where $n \geq 2m$, denote a set of n tasks sharing one resource ℓ_1 such that, for each T_i, $e_i = 1$, $N_{i,1} = 1$, $L_{i,1} = 1$, and $p_i = m$ if $i < m$, $p_i = mn/2$ if $m \leq i \leq 2m-2$, and $p_i = mn$ otherwise.

Lemma 11. *There exists an arrival sequence for $\tau^{prio}(n)$ such that $\max_{T_i \in \tau}\{b_i\} = \Omega(mn)$ (under s-aware analysis) when ordering requests by either non-decreasing job deadline or static priority under any JLSP scheduler.*

Proof. Without loss of generality, assume that n is an integer multiple of $2m$. We first consider partitioned scheduling and assume that $\tau^{prio}(n)$ is partitioned such that $P_i = i$ if $i < m$ and $P_i = m$ otherwise.

Consider the synchronous, periodic arrival sequence, *i.e.*, each $J_{i,j}$ is released at $a_{i,j} = (j-1) \cdot p_i$, and issues one request $\mathcal{R}_{i,j}$, where $\|\mathcal{R}_{i,j}\| = 1$. The resulting schedule for $m = 3$ and $n = 6$ is illustrated in Fig. 7.

Since the task set is serialized by ℓ_1, the order of job completions is fully determined by the queuing discipline under any work-conserving scheduler. Recall from Sec. 2 that tasks are indexed in order of decreasing priority, and that deadline ties are broken in favor of jobs of higher-indexed tasks. Thus, if requests are either EDF- or static-priority-ordered, then, by construction, $J_{n,1}$'s request is the last one to be satisfied at time $n \cdot m - 1$. By Def. 2, $J_{n,1}$ incurs pi-blocking whenever no higher-priority job is scheduled on P_m during $[0, n \cdot m - 1)$. By construction, P_m is used for only $n-1$ time units during $[0, n \cdot m - 1)$. Thus, $J_{n,1}$ is pi-blocked for at least $b_n \geq n \cdot m - 1 - (n-1) = (m-1) \cdot n = \Omega(mn)$ time units. Since at most one job is scheduled at any time, pi-blocking does not decrease under global scheduling. □

Total pi-blocking is similarly non-optimal in such cases.

Scheduler	Protocol	Queue	Maximum pi-blocking	
			s-oblivious	s-aware
global	FMLP [8]	FIFO	$\Theta(n)$	$\Theta(n)$
	PIP [14]	priority	$\Omega(n)$	$\Omega(mn)$
	PPCP [14]	priority	$\Omega(n)$	$\Omega(mn)$
	OMLP	hybrid	$\Theta(m)$	$\Omega(mn)$
partitioned	DPCP [22]	priority	$\Omega(n)$	$\Omega(mn)$
	MPCP [20]	priority	$\Omega(n)$	$\Omega(mn)$
	FMLP [8]	FIFO	$O(n^2)$	$O(n^2)$
	OMLP	hybrid	$\Theta(m)$	$\Omega(mn)$
	SPFP	FIFO	$\Theta(n)$	$\Theta(n)$

Table 1: Summary of results. Only the OMLP is optimal in the s-oblivious case because both FIFO- and priority-ordered queues give rise to $\Omega(n)$ s-oblivious pi-blocking (see Sec. 3.2). In the s-aware case, all protocols employing priority queues are subject to a lower bound of $\Omega(mn)$ (Lemma 11), and similar reasoning also applies to the OMLP. Recall from Sec. 2 that an upper bound on s-aware pi-blocking bounds s-oblivious pi-blocking as well. A comparison of each protocol's exact (*i.e.*, non-asymptotic) bounds requires empirical experiments since such bounds usually cannot be expressed as closed-form expressions and is thus beyond the scope of this paper.

Lemma 12. *There exist task systems with* $\sum_{i=1}^{n} b_i = \Omega(m(n-m)^2 + m^2)$ *(under s-aware analysis) when ordering requests by either non-decreasing job deadline or static priority under any JLSP scheduler.*

Proof. Omitted due to space constraints; can be shown analogously to Lemma 10 by considering a modified version of $\tau^{prio}(n)$ in which $p_i = m \cdot (n - m + 1)$ if $i \geq m$. The $-m$ and m^2 terms arise because $T_1,\ldots,T_{m-1}$ incur only little pi-blocking. □

Note that Lemmas 11 and 12 depend on $\max\{p_i\}/\min\{p_i\} = mn/m$ not being constant. If $\max\{p_i\}/\min\{p_i\}$ is constrained to be constant, then any (reasonable) protocol likely ensures $O(n)$ maximum pi-blocking if shared resources are not continuously in use.

5 Conclusion

We have presented precise definitions of pi-blocking for s-oblivious and s-aware multiprocessor schedulability analysis. Using these definitions, we have established a number of bounds on pi-blocking that are applicable to any JLFP global or partitioned scheduling algorithm. For the case of s-oblivious analysis, we have shown that $\Omega(m)$ worst-case pi-blocking is fundamental. We have also presented global and partitioned variants of a new asymptotically optimal locking protocol, the OMLP. Worst-case s-oblivious pi-blocking under the global (partitioned) OMLP is $O(m)$ under any global (partitioned) JLFP scheduler. The OMLP is not just of theoretical interest. For example, the only prior proposed locking protocol for G-EDF is the FMLP, and the OMLP has substantially better pi-blocking bounds than the FMLP.

For the case of s-aware analysis, we have shown that $O(m)$ worst-case pi-blocking is not possible by presenting a general $\Omega(n)$ lower bound. We have also shown that the global FMLP meets this bound and thus is asymptotically optimal. In the partitioned case, we have presented a simple FIFO locking protocol that also meets this bound. This algorithm achieves $O(n)$ worst-case pi-blocking by serializing requests of all resources. While such an approach may be of questionable practical utility, its existence nonetheless shows that our lower bound is tight. We have further shown that no locking protocol that orders lock requests on a static-priority or EDF basis can be optimal by establishing an $\Omega(mn)$ lower bound that is applicable to any such protocol. Our results are summarized in Table 1.

It is important to note that asymptotic optimality as a function of m or n does *not* imply that a locking protocol is the best to use in all circumstances. Obviously, asymptotic claims ignore constant factors. Additionally, a non-optimal algorithm could yield lower pi-blocking delays for some task systems (just like the non-optimal Quicksort algorithm is often faster than optimal sorting algorithms).

In future work, it would be interesting to empirically investigate such practical tradeoffs with schedulability and implementation studies. Likewise, as shown in Table 1, asymptotic upper bounds have yet to be derived for many existing protocols—finding bounds that are asymptotically tight may be non-trivial in some cases. With regard to locking optimality, we would like to more carefully examine the effects of certain parameters (like request lengths and the number of requests per job) that we have assumed to be constant. Similarly, bounds in the case where the maximum number of tasks sharing a resource is constant (*e.g.*, if any resource is accessed by at most three tasks) are of practical interest. Also, nesting with fine-grained locking (*i.e.*, not using group locks) and reader/writer locks warrant further attention. Finally, we note that current hard real-time G-EDF analysis is s-oblivious. S-aware analysis may enable resource sharing to be treated less pessimistically.

References

[1] T. Baker. Stack-based scheduling for realtime processes. *Real-Time Systems*, 3(1):67–99, 1991.

[2] T. Baker. Multiprocessor EDF and deadline monotonic schedulability analysis. *Proc. of the 24th IEEE Real-Time Systems Symposium*, pages 120–129, 2003.

[3] S. Baruah. Techniques for multiprocessor global schedulability analysis. In *Proc. of the 28th IEEE Real-Time Systems Symposium*, pages 119–128, 2007.

[4] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, and S. Stiller. Improved multiprocessor global schedulability analysis. *Real-Time Systems*, 46(1):3–24, 2010.

[5] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Proc. of the 28th IEEE Real-Time Systems Symposium*, pages 149–160, 2007.

[6] M. Bertogna, M. Cirinei, and G. Lipari. Improved schedulability analysis of EDF on multiprocessor platforms. In *Proc. of the 17th Euromicro Conference on Real-Time Systems*, pages 209–218, 2005.

[7] M. Bertogna, M. Cirinei, and G. Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Trans. on Parallel and Distributed Systems*, 20(4):553–566, 2009.

[8] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proc. of the 13th IEEE Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–57, 2007.
[9] B. Brandenburg and J. Anderson. An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS$^{\mathrm{RT}}$. In *Proc. of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 185–194, 2008.
[10] B. Brandenburg and J. Anderson. Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-Time Systems*, 46(1):25–87, 2010.
[11] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Chapman and Hall/CRC, 2004.
[12] C. Chen and S. Tripathi. Multiprocessor priority ceiling based protocols. Technical Report CS-TR-3252, Univ. of Maryland, 1994.
[13] U. Devi, H. Leontyev, and J. Anderson. Efficient synchronization under global EDF scheduling on multiprocessors. In *Proc. of the 18th Euromicro Conference on Real-Time Systems*, pages 75–84, July 2006.
[14] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *Proc. of the 30th IEEE Real-Time Systems Symposium*, pages 377–386, 2009.
[15] P. Gai, M. di Natale, G. Lipari, A. Ferrari, C.Gabellini, and P. Marceca. A comparison of MPCP and MSRP when sharing resources in the Janus multiple processor on a chip platform. In *Proc. of the 9th IEEE Real-Time And Embedded Technology Application Symposium*, pages 189–198, 2003.
[16] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2-3):187–205, 2003.
[17] K. Lakshmanan, D. Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Proc. of the 30th IEEE Real-Time Systems Symposium*, pages 469–478, 2009.
[18] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 30:46–61, 1973.
[19] J. Lopez, J. Diaz, and D. Garcia. Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28(1):39–68, 2004.
[20] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. *Proc. of the 10th International Conference on Distributed Computing Systems*, pages 116–123, 1990.
[21] R. Rajkumar. *Synchronization In Real-Time Systems – A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.
[22] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. *Proc. of the 9th Real-Time Systems Symposium*, pages 259–269, 1988.
[23] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Trans. on Computers*, 39(9):1175–1185, 1990.
[24] J. Strosnider, J. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. on Computers*, 44(1):73 –91, 1995.

A Improved Bounds

As noted in Sec. 3.2.2, the bound given in Lemma 4 (and, respectively, in Lemma 7 under partitioning) overestimate worst-case pi-blocking incurred when requesting resources that are not heavily contended: if requests are infrequent and mostly short, then assuming that that all queues are "saturated" with the longest-possible request is needlessly pessimistic. This is best illustrated with an example.

Ex. 1. Consider three tasks T_1, T_2, and T_3 with parameters as indicated in Table 2 sharing one resource ℓ_1. Suppose that

Task	e_i	p_i	$N_{i,1}$	$L_{i,1}$
T_1	9	50	2	1
T_2	6	30	1	3
T_3	3	20	1	1

Table 2: Example tasks set. Three tasks T_1, T_2, T_3 sharing one resource ℓ_1.

these tasks are scheduled on $m = 16$ processors (together with a number of other tasks that do not access ℓ_1). Under the global OMLP, by Lemma 4, a job J_3 is pi-blocked for at most $b_3 = N_{3,1} \cdot 2 \cdot (m-1) \cdot \max\{L_{1,1}, L_{2,1}, L_{3,1}\} = 1 \cdot 2 \cdot 15 \cdot 3 = 90$ time units. Given that only jobs of T_2 issue requests of length 3, this clearly overestimates actual worst-case pi-blocking. Based on this coarse-grained bound, T_3 would be (wrongly) deemed unschedulable since $b_3 > p_3$.

In this appendix, we derive less-pessimistic, albeit notationally more tedious, bounds for s-oblivious pi-blocking under the global and partitioned OMLP that better reflect task periods and per-task maximum request lengths.

A.1 S-Oblivious Pi-Blocking under the Global OMLP

Lemma 4 can be improved by deriving a better approximation of the set of requests that can delay a job in the worst case. Intuitively, the idea is to "count" how many times each task T_x can request a shared resource while J_i is pending, and to charge each $L_{x,k}$ individually based on these counts.

Ex. 2. Continuing Ex. 1, we derive the "worst-case interference" for J_3, which we then use to bound maximum pi-blocking. Because $p_3 < p_2 < p_1$, J_3 can overlap (*i.e.*, be pending concurrently) with at most two jobs of T_1 and T_2 each. Each J_1 can request ℓ_1 twice, and each J_2 can request ℓ_1 once. Thus, at most six requests $(4 \times T_1, 2 \times T_2)$ can interfere with J_3 in the worst case. Since $L_{1,1} = 1$, this method yields a much tighter upper bound of $b_3 = 4 \cdot L_{1,1} + 2 \cdot L_{2,1} = 4 \cdot 1 + 2 \cdot 3 = 10$.

To formalize this approach, we require a safe approximation of the set of possibly-interfering requests issued by jobs of each competing task T_x. As illustrated in Ex. 2, this requires a bound on the maximum number of jobs of T_x that may execute (and thus issue requests) concurrently with J_i.

Lemma 13 (from [10])**.** *At most $\lceil (t + r_x)/p_x \rceil$ jobs of a task T_x can execute during any interval of length t.*

By the definition of $N_{x,k}$, Lemma 13 implies that jobs of T_x issue at most $\lceil (t + r_x)/p_x \rceil \cdot N_{x,k}$ requests for ℓ_k over any interval of length t. This yields the following definition.

Def. 5. The *worst-case task interference* generated by jobs of T_x over any interval of length t is the set of requests

$$\mathit{tif}(T_x, \ell_k, t) \triangleq \{\mathcal{R}_{x,y} \mid 1 \le y \le N_{x,j} \cdot \lceil (t + r_x)/p_x \rceil\},$$

where $\|\mathcal{R}_{x,y}\| = L_{x,k}$ for each $\mathcal{R}_{x,y}$.

Def. 5 characterizes the worst-case demand for ℓ_k by jobs of T_x, *i.e.*, it is a safe upper bound of both the number of requests issued by T_x as well as their respective lengths.

Ex. 3. Suppose $r_i = p_i$ for each T_i (see Sec. A.3 below). Continuing Ex. 2, let $t = p_3 = 20$. Then $\mathit{tif}(T_1, \ell_1, 20) = \{\mathcal{R}_{1,1}, \mathcal{R}_{1,2}, \mathcal{R}_{1,3}, \mathcal{R}_{1,4}\}$, where $\|\mathcal{R}_{1,1}\| = \|\mathcal{R}_{1,2}\| = \|\mathcal{R}_{1,3}\| = \|\mathcal{R}_{1,4}\| = 1$, since $\lceil(20 + p_1)/p_1\rceil = 2$ and $N_{1,1} = 2$. Similarly, $\mathit{tif}(T_2, \ell_1, 20) = \{\mathcal{R}_{2,1}, \mathcal{R}_{2,2}\}$, where $\|\mathcal{R}_{2,1}\| = \|\mathcal{R}_{2,2}\| = 3$.

Task interference bounds contention due to a single task. We similarly define interference from a subset of τ.

Def. 6. For a set of tasks S, we let

$$\mathit{xif}(S, \ell_k, t) \triangleq \bigcup_{T_x \in S} \mathit{tif}(T_x, \ell_k, t)$$

denote the *worst-case request interference*, and, for each v, $1 \le v \le |\mathit{xif}(S, \ell_k, t)|$, let $\mathit{xif}_v(S, \ell_k, t)$ denote the vth longest request in $\mathit{xif}(S, \ell_k, t)$ (with ties broken arbitrarily).

Ex. 4. Continuing Ex. 3, let $S = \{T_1, T_2\}$ and $t = 20$. Then $\mathit{xif}(S, \ell_1, 20) = \{\mathcal{R}_{2,1}, \mathcal{R}_{2,2}, \mathcal{R}_{1,1}, \mathcal{R}_{1,2}, \mathcal{R}_{1,3}, \mathcal{R}_{1,4}\}$ in order of non-increasing length, *i.e.*, $\|\mathit{xif}_1(S, \ell_1, 20)\| = 3$, $\|\mathit{xif}_2(S, \ell_1, 20)\| = 3$, $\|\mathit{xif}_3(S, \ell_1, 20)\| = 1$, *etc.* Thus, the bound that was manually derived in Ex. 2 can be expressed as $b_3 = \sum_{v=1}^{6} \|\mathit{xif}_v(S, \ell_1, p_3)\| = 10$.

We formalize this bound next on a per-resource basis. We provide a per-resource bound in anticipation of a later refinement for special cases (Lemma 15 below).

Def. 7. Let $b_{i,k}$ denote a bound on the total pi-blocking incurred by J_i due to requests for resource ℓ_k. Under the global OMLP, $b_i = \sum_{k=1}^{q} b_{i,k}$.

Combining the worst-case interference (Def. 6) with Lemma 3 yields the following more accurate (but asymptotically unchanged) bound.

Lemma 14. *Let $S = \tau \setminus \{T_i\}$, and let $\alpha_k = \min(N_{i,k} \cdot 2 \cdot (m-1), |\mathit{xif}(S, \ell_k, r_i)|)$. J_i is pi-blocked due to requests for ℓ_k for at most $b_{i,k} \triangleq \sum_{v=1}^{\alpha_k} \|\mathit{xif}_v(S, \ell_k, r_i)\|$.*

Proof. Follows from Lemma 3 and Def. 6: total pi-blocking does not exceed the length of the $N_{i,k} \cdot 2 \cdot (m-1)$ longest interfering requests (if that many exist, hence α_k). □

Deriving the worst-case request interference avoids over-counting long-but-infrequent requests. However, if a resource is shared by at most m tasks, then the above bound fails to fully reflect the strict FIFO-ordering of jobs in FQ_k.

Ex. 5. Consider the task set from Ex. 1. Since only two other tasks access ℓ_1 and $m = 16$, J_3 is guaranteed to immediately enter FQ_1 when it requests ℓ_1. Thus, J_3 has to await the completion of at most one request of T_1 and one request of T_2 since jobs are FIFO-ordered in FQ_1 and at most one job per task is pending at any time. This implies a tighter bound of $b_3 = N_{3,1} \cdot L_{1,1} + N_{3,1} \cdot L_{2,1} = 1 \cdot 1 + 1 \cdot 3 = 4$.

This improvement is formalized next.

Def. 8. Let A_k denote the number of tasks accessing ℓ_k, *i.e.*, $A_k \triangleq |\{T_i \mid 1 \le i \le n \wedge N_{i,k} > 0\}|$.

Lemma 15. *Let $C_{x,k} = |\mathit{tif}(T_x, \ell_k, r_i)|$ denote the maximum number of times that jobs of T_x request ℓ_k while J_i is pending. If $A_k \le m$, then J_i is pi-blocked due to requests for ℓ_k for at most $b_{i,k} \triangleq \sum_{x=1; x \ne i}^{n} \min(N_{i,k}, C_{x,k}) \cdot L_{x,k}$.*

Proof. If $A_k \le m$, then J_i never enters PQ_k. Due to the FIFO ordering in FQ_k, J_i is pi-blocked by at most one request from $A_k - 1$ other tasks each time that it requests ℓ_k, for a total of at most $N_{i,k}$ requests per task. However, if jobs of a competing task T_x issue fewer than $N_{i,k}$ requests, then not each request of J_i is blocked by a request of T_x. Hence, J_i is pi-blocked for a total duration of at most $\min(N_{i,k}, C_{x,k}) \cdot L_{x,k}$ for each T_x (note that $C_{x,k} = 0$ if $N_{x,k} = 0$). □

Thus, J_i is pi-blocked in total for at most $b_i = \sum_{k=1}^{q} b_{i,q}$, with each $b_{i,q}$ defined as given in Lemma 15 if $A_k \le m$, and defined as given in Lemma 14 otherwise.

A.2 S-Oblivious Pi-Blocking under the Part. OMLP

The same approach, namely to derive a more accurate approximation of the worst-case request interference, can be easily transferred to the partitioned OMLP. In particular, Lemma 7 overestimates the contention arising on each remote processor if resources are requested only infrequently. Based on Def. 6, we can state the following more-accurate bound that takes worst-case interference into account.

Lemma 16. *While holding CT_{P_i}, J_i is pi-blocked for at most*

$$B_i^{\mathit{fifo}} \triangleq \sum_{k=1}^{q} \sum_{\substack{o=1 \\ o \ne P_i}}^{m} \sum_{v=1}^{\alpha_{k,o}} \|\mathit{xif}_v(\mathit{part}(o), \ell_k, r_i)\|,$$

where $\alpha_{k,o} = \min(N_{i,k}, |\mathit{xif}(\mathit{part}(o), \ell_k, r_i)|)$.

Proof. Each time that J_i requests a resource ℓ_k, J_i must wait for the completion of at most one request originating from each remote processor. Hence, it is sufficient to consider the $N_{i,k}$ longest requests for ℓ_k originating from each remote processor (if that many exist, hence $\alpha_{k,o}$). □

It is further possible to improve Lemmas 6 and 8 by considering only resources accessed by lower-priority jobs. This, however, requires scheduling-algorithm-specific analysis. Note that Lemma 16 does not depend on the employed scheduling algorithm.

A.3 Computing b_i

Both $\mathit{tif}(T_i, \ell_k, t)$ and $\mathit{xif}(S, \ell_k, r_i)$ (Defs. 5 and 6) depend on each T_i's worst-case response time r_i, which in turn depends on the worst-case acquisition delay and thus worst-case interference. This circular dependency can be resolved either by conducting a fixed-point search for an upper bound on r_i (a similar approach is used to analyze the MPCP [17]), or by simply substituting p_i for r_i, which is a safe approximation if the resulting τ' is schedulable. The former requires repeated response time approximations (*e.g.*, see [7]), whereas the latter can be computed immediately, albeit at the cost of increased pessimism.