The OMLP Family of Optimal Multiprocessor Real-Time Locking Protocols

Björn B. Brandenburg · James H. Anderson

Abstract This paper presents the first suspsension-based multiprocessor real-time locking protocols with asymptotically optimal blocking bounds (under certain analysis assumptions). These protocols can be applied under any global, clustered, or partitioned job-level fixed-priority scheduler and support mutual exclusion, reader-writer exclusion, and k-exclusion constraints. Notably, the reader-writer and k-exclusion protocols are the first analytically-sound suspension-based multiprocessor real-time locking protocols of their kind. To formalize a notion of "optimal blocking," precise definitions of what constitutes "blocking" in a multiprocessor real-time system are given and a simple complexity metric for real-time locking protocols, called maximum priority-inversion blocking (pi-blocking), is introduced. It is shown that, in a system with m processors, $\Omega(m)$ maximum pi-blocking is unavoidable. This bound is shown to be asymptotically tight with the introduction of the $O(m)$ multiprocessor locking protocol (OMLP) family presented herein, which includes protocols that ensure an upper bound on maximum pi-blocking that is approximately within a factor of two of the lower bound. In addition to the coarse-grained asymptotic bounds, detailed blocking bounds suitable for schedulability analysis are derived using holistic blocking analysis. Based on the detailed bounds, the proposed locking protocols are compared with each other and with previously-proposed protocols in an empirical schedulability study involving more than one billion task sets. In this study, the OMLP was found to perform better than two variants of the classic (but non-optimal) multiprocessor priority-ceiling protocol (MPCP).

Keywords real-time synchronization · priority inversion · optimal locking protocol · mutual exclusion · reader-writer exclusion · k-exclusion · schedulability study

1 Introduction

When semaphores are used to coordinate access to shared resources such as I/O devices or shared data structures, some blocking among tasks is unavoidable. In real-time systems, such blocking gives rise to *priority inversions*, which, intuitively, occur when a high-priority task must wait for a low-priority one. As this can endanger temporal correctness, a real-time locking protocol is required to bound the maximum duration of priority inversion. In this paper, we present the first such protocols for multiprocessors that are provably optimal (within a factor of the lower bound that approaches two or four, depending on the protocol).

Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS)
Campus E 1 5, D-66123 Saarbrücken, Germany
E-mail: bbb@mpi-sws.org

James H. Anderson
The University of North Carolina at Chapel Hill (UNC-CH)
Department of Computer Science CB# 3175, Chapel Hill, NC 27599-3175, USA
E-mail: anderson@cs.unc.edu

There are two approaches to realizing locks in multiprocessor systems: in *spin-based* (or *spinlock*) protocols, jobs wait for resources by executing a delay loop, and in *suspension-based* (or *semaphore*) protocols, waiting jobs relinquish their processor. In principle, suspension-based protocols are preferable because waiting jobs waste processor cycles under spin-based protocols. In practice, spin-based protocols benefit from low overheads (compared to the cost of suspending and resuming tasks), so that spinning can in fact be preferable if all critical sections are short, that is, if tasks use resources for at most a few microseconds [14, 15, 20]. Nonetheless, suspension-based protocols are still needed to support shared resources that inherently cause critical sections to be long (*e.g.*, stable storage), as spinning would result in substantial wastage in such cases. In this paper, we focus on suspension-based protocols.

Clearly, an "optimal" real-time locking protocol should minimize blocking to the extent possible. However, while optimal uniprocessor real-time locking protocols have long been known [43, 47], no provably optimal suspension-based multiprocessor real-time locking protocols have been proposed to date. In fact, prior work (reviewed below) did not provide a general, *precise* definition of what constitutes "blocking" in a multiprocessor real-time system. Rather, the design goal of "minimal blocking" was only informally understood and existing protocols have been analyzed by providing upper bounds on lock-acquisition delays that would be sufficient under any reasonable definition of blocking. We close this gap by showing that there are actually two different notions of priority inversion, called *suspension-oblivious* and *suspension-aware*, that arise due to differences in how task suspensions (which are notoriously hard to analyze [45]) are handled in existing schedulability tests (Section 3.1). Intuitively, suspension-aware analysis allows task suspensions to be accounted for explicitly, whereas suspension-oblivious analysis requires such suspensions to be modeled as computation instead. Based on these definitions, we propose *maximum priority inversion blocking* (*maximum pi-blocking*) as a natural complexity metric for locking protocols (Section 3.2). Notably, suspension-oblivious and suspension-aware schedulability analysis yield different lower bounds on maximum pi-blocking.

In this paper, we focus on locking protocols for suspension-oblivious analysis. We show that, in a multiprocessor system with m processors under suspension-oblivious analysis, $\Omega(m)$ maximum pi-blocking is unavoidable in the general case (Section 3.3). This bound is asymptotically tight, which we show by introducing a family of $O(m)$ locking protocols (OMLP) in Section 4. The OMLP family includes two *mutual exclusion* (or *mutex*) protocols for serially-reusable resources, a protocol for *reader-writer* (RW) exclusion, where only updates must be exclusive and reads may overlap with each other, and a protocol for k*-exclusion* constraints, where there are k replicas of a resource and tasks require exclusive access to any one replica. Notably, the latter two are the first suspension-based multiprocessor real-time locking protocols for RW and k-exclusion. We prove the OMLP to be asymptotically optimal under suspension-oblivious analysis: the mutex protocols and the k-exclusion protocol ensure a bound on maximum pi-blocking that is approximately within a factor of two of the lower bound, and the RW protocol's bound is within a factor of four of the lower bound (Section 4.6). In addition to being asymptotically optimal, the OMLP is also of considerable practical interest because it compares favorably with previously-proposed protocols for both suspension-aware and suspension-oblivious analysis (Section 5), and because it can be applied under a wide range of multiprocessor real-time schedulers.

With regard to the latter, a real-time locking protocol must be tightly integrated with the scheduler since real-time tasks typically require (some) processor service while using shared resources (*e.g.*, this is the case when accessing shared data structures or when executing driver routines to access an I/O device). Most prior work on multiprocessor real-time locking protocols has focused on either *partitioned* scheduling (where tasks are statically assigned to processors and each processor is scheduled individually) or *global* scheduling (where all processors serve a single ready queue and tasks may migrate freely). Because partitioning requires a bin-packing-like task assignment problem to be solved, global scheduling offers some theoretical advantages over partitioning, but does so at the expense of higher runtime costs. *Clustered* scheduling [5, 21] is an attractive compromise between (or generalization of) the two extremes, where tasks are partitioned onto disjoint clusters of cores and a global scheduling policy is used within each cluster. Clustered scheduling simplifies the task assignment problem (there are fewer and larger bins) and incurs less overhead than global scheduling (by aligning clusters with the underlying hardware topology). Recent experimental work has confirmed the effectiveness of clustered scheduling on large multicore, multi-chip platforms [9, 10, 14] and we expect clustered scheduling to grow in importance as multicore platforms become larger and less uniform.

However, clustered scheduling poses significant challenges from a locking point of view, and there is only scant support for clustered scheduling among prior locking protocols. Since clustered scheduling combines aspects from both global and partitioned scheduling, the established mechanisms for bounding priority inversions—*priority inheritance* under global scheduling and *priority boosting* under partitioned scheduling—do not transfer to clustered scheduling (Section 4.1). To overcome this limitation, we introduce "priority donation," a new mechanism to bound priority inversions. Using priority donation as a basis, all but one of protocols in the OMLP family support any global, partitioned, or clustered *job-level fixed-priority* (JLFP) scheduler [23], a large class of schedulers that includes the commonly-used *fixed-priority* (FP) and *earliest-deadline first* (EDF) policies. As discussed next, these are the first suspension-based multiprocessor real-time locking protocols designed specifically for clustered scheduling.

Related work Most prior work has been directed at EDF and FP scheduling and their partitioned and global multiprocessor extensions (denoted as P-FP, P-EDF, G-FP, and G-EDF, respectively). On uniprocessors, real-time locking is well-understood. The classic uniprocessor *stack resource policy* (SRP) [3] and the *priority ceiling protocol* (PCP) [25, 43, 47] both support *multi-unit resources*, which is a generalized resource model that can be used to realize mutex, RW, and k-exclusion constraints. In the case of mutual exclusion, both protocols limit the maximum duration of priority inversion to the length of one (outermost) critical section, which is arguably optimal.

Work on multiprocessor protocols has mostly focused on mutex constraints to date. The first such protocols were proposed by Rajkumar *et al*. [42–44], who designed two suspension-based PCP extensions for P-FP-scheduled systems, the *distributed* and the *multiprocessor priority ceiling protocol* (DPCP and MPCP, respectively). In later work, improved analysis of the MPCP incorporating uniprocessor response-time analysis [2, 35] was independently developed by both Schliecker *et al*. [46] and Lakshmanan *et al*. [36]; Lakshmanan *et al*. further proposed a variant of the MPCP for suspension-oblivious analysis and a partitioning heuristic [36]. Recently, Hsiu *et al*. [34] studied the problem of finding optimal and near-optimal task and resource assignments in distributed systems employing a protocol similar to the DPCP.

In early work on P-EDF-scheduled systems, suspension- and spin-based protocols were presented by Chen and Tripathi [24] and Gai *et al*. [32]. Devi *et al*. [28] presented an analysis of spinlocks under G-EDF scheduling. Block *et al*. [13] presented the *flexible multiprocessor locking protocol* (FMLP), which can be used under G-EDF, P-EDF, and P-FP [16] scheduling, supports both spinlocks and semaphores, and generalizes Gai *et al*.'s and Devi *et al*.'s protocols.

More recently, Easwaran and Andersson [29] considered suspension-based protocols for G-FP-scheduled systems. They presented an analysis of the *priority inheritance protocol* (PIP) and proposed the *parallel priority-ceiling protocol* (PPCP). Andersson and Easwaran [1] also designed a multiprocessor locking protocol with a bounded resource augmentation factor.[1] Extending Easwaran and Andersson's work [29], Macariu and Cretu [39] proposed an extension of the PIP under G-FP scheduling that limits the extent of priority inversions caused by resource-holding lower-priority jobs with raised effective priorities.

In work aimed at maintaining temporal isolation among tasks in mixed real-time/non-real-time environments, Faggioli *et al*. [31] presented the *multiprocessor bandwidth-inheritance protocol* (MBWI), a scheduler-agnostic locking protocol under which tasks wait both by spinning and by suspending. Because it is scheduler agnostic, the MBWI can be applied to clustered scheduling, but, in contrast to the OMLP, actual spinning takes place under the MBWI. Finally, Nemati *et al*. [40] proposed a suspension-based locking protocol for partitioned scheduling that facilitates the integration of independently-developed, opaque application components consisting of multiple tasks each by abstracting their joint resource requirements into interface specifications.

To the best of our knowledge, none of the cited suspension-based real-time locking protocols has been analyzed under clustered scheduling. Further, even under the schedulers for which they were designed, none of the cited suspension-based real-time locking protocols ensures asymptotically optimal maximum pi-blocking, irrespective of whether the underlying schedulability analysis is suspension-aware or suspension-oblivious.

[1] A protocol has a *resource augmentation factor* x if any feasible task set that is not schedulable under it is guaranteed to be schedulable on an x-times faster processor.

In prior work on spin-based locking protocols [18], we presented the first spin-based real-time multiprocessor RW protocol. We showed that existing non-real-time RW locks are undesirable for real-time systems and proposed *phase-fair* RW locks, under which readers incur only constant delays, as an alternative. In recent work, the analysis of mutex and RW spinlocks under JLFP schedulers has been extended to clustered scheduling [14]. To the best of our knowledge, suspension-based RW and k-exclusion protocols have not been considered in prior work on real-time multiprocessors. While PCP variants could conceivably be used under partitioned scheduling, we are not aware of relevant analysis.

The material presented in this paper extends two prior conference papers [17, 19]. In particular, we

- have added a detailed derivation and discussion of the constant factors in the OMLP's blocking bounds (Section 4.6),
- report on new large-scale schedulability experiments involving more than one billion task sets (Section 5),
- have developed a new objective methodology for reporting schedulability results based on bootstrap confidence intervals (Section 5.1.3),
- discuss in detail when each protocol is most appropriate (Sections 5.2–5.5), and
- present detailed (*i.e.*, non-asymptotic) blocking analysis suitable for schedulability analysis for each of the proposed locking protocols using a holistic blocking analysis framework (Appendix A).

We begin by providing needed background and definitions.

2 System Model

We consider the problem of scheduling a set of n sporadic tasks $\tau = \{T_1, \ldots, T_n\}$ on $m \geq 2$ identical 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, where T_i's *utilization* u_i is given by the ratio $u_i = e_i/p_i$. $J_{i,j}$ 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, where successive releases are seperated by at least p_i time units ($a_{i,j+1} \geq a_{i,j} + p_i$).

Task T_i's *maximum response time* r_i denotes the maximum time that any $J_{i,j}$ remains pending. Task T_i is *schedulable* if it can be shown that $r_i \leq p_i$, that is, if each $J_{i,j}$ completes within p_i time units of its release. Note that we assume implicit deadlines only for the sake of simplicity; the presented results do not depend on the choice of deadline constraint. We omit the job index j if it is irrelevant and let J_i denote an arbitrary job.

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. Pending jobs are presumed ready unless suspended by a locking protocol. We consider the effect of allowing locking-unrelated suspensions in Section 4.6.

2.1 Scheduling

Under *clustered scheduling* [5, 21], processors are grouped into $\frac{m}{c}$ non-overlapping sets (or *clusters*) of c processors each, which we denote as $C_1, \ldots, C_{\frac{m}{c}}$.[2] *Global* and *partitioned* scheduling are special cases of clustered scheduling, where $c = m$ and $c = 1$, respectively. Each task is statically assigned to a cluster. Jobs may migrate freely within clusters, but not across cluster boundaries.

When bounding locking-related delays, it is often required to consider the subset of jobs assigned to a particular cluster. We let τ_k denote the set of tasks assigned to the k^{th} cluster, and let P_i denote the cluster (or partition) to which T_i is assigned. A task T_l is *local* to task T_i if $P_l = P_i$, and *remote* otherwise.

We assume that, within each cluster, jobs are scheduled from a single ready queue using a work-conserving *job-level fixed-priority* (JLFP) policy [23]. A JLFP policy assigns each job a fixed *base priority*, but a job's *effective priority* may temporarily exceed its base priority when raised by a locking protocol (see below). Within

[2] Without loss of generality, we assume uniform cluster sizes and $\frac{m}{c} \in \mathbb{N}$. Non-uniform cluster sizes could be trivially integrated into the presented analysis at the expense of additional notation.

each cluster, at any point in time, the c ready jobs (if that many exist) with the highest effective priorities are scheduled. We assume that ties in priority are broken in favor of lower-indexed tasks (*i.e.*, priorities are unique).

We consider *global, partitioned*, and *clustered* EDF (G-EDF, P-EDF, and C-EDF, respectively) as representative algorithms of the class of JLFP policies.

2.2 Resource Model

We consider three types of shared resources that differ with respect to their sharing constraint. *Mutual exclusion* of accesses is required for *serially-reusable* resources, which may be used by at most one job at any time. *Reader-writer exclusion* (RW exclusion) [26] is sufficient if a resource's state can be observed without affecting it: only *writes* (*i.e.*, state changes) are exclusive and multiple *reads* may be satisfied simultaneously. Resources of which there are k identical replicas are subject to a *k-exclusion* constraint: each replica is only serially reusable and thus requires mutual exclusion, but up to k requests may be satisfied at the same time by delegating them to different replicas.

Mutex constraints are most common in practice. However, the need for RW synchronization arises naturally in many situations, too. Two common examples are few-producers/many-consumers relationships (*e.g.*, obtaining and distributing sensor data) and rarely changing shared state (*e.g.*, configuration information). Of the three constraints, we expect k-exclusion constraints to be the least common. However, k-exclusion is required whenever there are multiple identical co-processors. For example, a system might contain multiple *graphics processing units* (GPUs) or *digital signal processors* (DSPs). Theoretically, one could further consider replicated resources with RW constraints, but we are not aware of any practical applications where such constraints arise and do not consider this combination of constraints.

We formalize resource sharing among sporadic tasks as follows. The system contains n_r shared resources $\ell_1, \ldots, \ell_{n_r}$ (such as shared data structures and I/O devices) besides the m processors. When a job J_i requires a resource ℓ_q it *issues* a *resource request* for ℓ_q. We let $\mathcal{R}_{i,q,v}$ denote the v^{th} resource request by task T_i for resource ℓ_q, where $v \geq 1$. In the case of RW constraints, we analogously let $\mathcal{R}^R_{i,q,v}$ and $\mathcal{R}^W_{i,q,v}$ denote the v^{th} read and write request for ℓ_q, respectively.

A request $\mathcal{R}_{i,q,v}$ is *satisfied* as soon as J_i *holds* ℓ_q, and *completes* when J_i *releases* ℓ_q. The *request length* is the time that J_i must execute before it releases ℓ_q.[3] We let $\mathcal{L}_{i,q,v}$ denote the request length of $\mathcal{R}_{i,q,v}$, let $N_{i,q}$ denote the maximum number of times that any J_i requests ℓ_q, and let $L_{i,q}$ denote the maximum length of such a request (*i.e.*, $\mathcal{L}_{i,q,v} \leq L_{i,q}$ for each $\mathcal{R}_{i,q,v}$), where $L_{i,q} = 0$ if $N_{i,q} = 0$. A task is *independent* if it does not require any shared resources. In the case of RW constraints, we analogously define $N^R_{i,q}$, $N^W_{i,q}$, $L^R_{i,q}$, and $L^W_{i,q}$ with respect to read and write requests, where $N_{i,q} = N^R_{i,q} + N^W_{i,q}$ and $L_{i,q} = \max(L^W_{i,q}, L^R_{i,q})$.

We assume that jobs request or hold at most one resource at any time and that tasks do not hold resources across job boundaries. Nesting could be supported with group locks as in the FMLP [13, 14], albeit at the expense of reduced parallelism.

2.3 Locking Protocols

To enforce sharing constraints, the operating system employs a *locking protocol* to order conflicting requests. If a request $\mathcal{R}_{i,q,v}$ of a job J_i cannot be satisfied immediately, then J_i incurs *acquisition delay* and cannot proceed with its computation while it waits for $\mathcal{R}_{i,q,v}$ to be satisfied. In this paper, we focus on protocols in which waiting jobs relinquish their processor and suspend. The *request span* of $\mathcal{R}_{i,q,v}$ starts when $\mathcal{R}_{i,q,v}$ is issued and lasts until it completes, that is, it includes the request length and any acquisition delay.

Locking protocols may temporarily raise a job's effective priority. Under *priority inheritance* [43, 47], the effective priority of a job J_i holding a resource ℓ_q is the maximum of J_i's priority and the priorities of all jobs

[3] For the sake of simplicity, we assume that jobs require a processor for the entirety of each critical section. This is accurate for shared data structures, but may be somewhat pessimistic when accessing devices. The assumption could be relaxed at the expense of additional notation by splitting each request length parameter into a processor component and a suspension component.

waiting for ℓ_q. Alternatively, under *priority boosting* [16, 17, 36, 42–44], a resource-holding job's priority is unconditionally elevated above the highest-possible base (*i.e.*, non-boosted) priority to expedite the completion of requests.

2.4 Priority Inversion and Blocking

The main goal in the design of real-time locking protocols is to minimize the worst-case duration of *priority inversions*. A priority inversion occurs when a job that *should* be scheduled (according to its base priority) is *not* scheduled (*i.e.*, either when a lower-priority job is scheduled instead or when a processor in its assigned cluster is idle). Priority inversions are problematic because they delay a job's completion and hence must be bounded and accounted for during schedulability analysis. It is important to note that acquisition delay and priority inversion are two different, although closely related, concepts: if a suspended job would not have been scheduled anyway due to the presence of higher-priority jobs, then there is no priority inversion. This matches the intuition that high-priority jobs should be granted access to contended resources sooner than lower-priority jobs.

In the real-time literature, acquisition delay that coincides with a priority inversion is traditionally called "blocking," whereas acquisition delay that does not coincide with a priority inversion lacks an established name (since it is irrelevant for analysis purposes). We avoid the term "blocking" because it is overloaded. In a real-time context, many other sources of schedulability-relevant delays are also commonly labeled "blocking," even if they do not coincide with a priority inversion. For example, release jitter and deferred execution are causes of "blocking" without priority inversion [38]. In the (non-real-time) synchronization literature, "blocking" is simply a synonym for acquisition delay. To further confuse matters, in an OS context, "blocking" is often used as a synonym for "suspending," which is not the same as the intended interpretation: in suspension-based locking protocols, the length of a suspension corresponds to the incurred acquisition delay, but not necessarily to the duration of priority inversion.

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

3 Blocking Optimality

In the uniprocessor case, locking protocols that ensure a provably optimal upper bound on pi-blocking have long been known. Indeed, under both the PCP [43, 47] and the SRP [3], jobs incur pi-blocking for the duration of at most one (outermost) critical section, which is obviously asymptotically optimal.

In the multiprocessor case, however, the question of "blocking optimality" had not received much, if any, attention. In fact, general, *precise* definitions of what actually constitutes "blocking" had not been formalized prior to our work. Rather, existing protocols have been analyzed using informally defined notions of blocking; to the effect that different locking protocols were analyzed using different assumptions. Without a precise definition of blocking, we clearly have no understanding of what constitutes *optimal* pi-blocking on multiprocessors.

Motivated by these considerations, we next formalize two notions of pi-blocking and define a notion of "blocking complexity," which we then use to establish the optimality of the protocols presented in this paper.

3.1 Priority Inversions in Multiprocessor Systems

The need for two notions of pi-blocking arises because multiprocessor schedulability analysis has not yet matured to the point that suspensions can be efficiently analyzed under all schedulers. In particular, most of the major G-EDF hard real-time schedulability tests do not inherently account for self-suspensions. Such analysis is *suspension-oblivious* (*s-oblivious*): jobs may suspend, but each e_i must be inflated by b_i prior

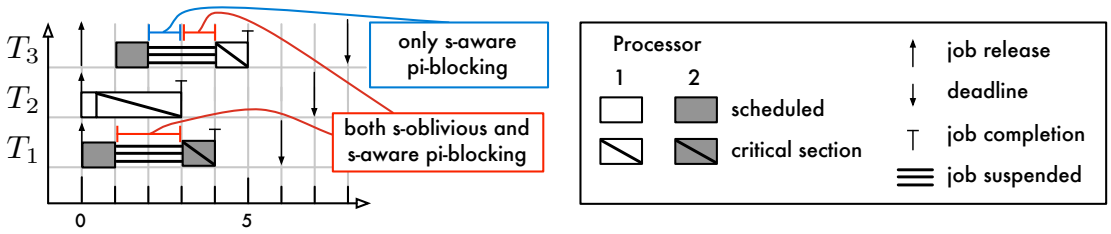


Fig. 1 S-oblivious and s-aware pi-blocking in a G-EDF schedule of three jobs sharing one resource on $m = c = 2$ processors

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 higher-priority 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 FP, P-FP, and, to some extent, for G-FP scheduling [2, 29, 36, 43]. 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 higher-priority jobs are counted as demand under s-oblivious analysis—the maximum time of priority inversion of each such job is included in its execution requirement e_i—the mere *existence* of c pending higher-priority jobs (in J_i's cluster) rules out a priority inversion. In contrast, under s-aware schedulability analysis only *ready* higher-priority jobs can nullify a priority inversion (since suspension times are not included in e_i).

The difference in what constitutes a priority inversion leads to two notions of pi-blocking. Since schedulability tests are applied on a cluster-by-cluster basis, pi-blocking is defined in both cases with respect to the tasks in each cluster. Recall from Section 2 that P_i denotes the cluster that T_i has been assigned to, and that τ_{P_i} denotes the set of tasks assigned to cluster P_i.

Definition 1 Under **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 c higher-priority jobs of tasks in τ_{P_i} are **pending**.

Definition 2 Under **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 c higher-priority ready jobs of tasks in τ_{P_i} are **scheduled**.

In both cases, "higher-priority" is interpreted with respect to base priorities. Notice that Definition 1 is weaker than Definition 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.

Example 1 The difference between s-oblivious and s-aware pi-blocking is illustrated in Figure 1, which shows a G-EDF schedule of three jobs sharing one resource. Job J_1 suffers acquisition delay during $[1, 3)$, and since no higher-priority jobs exist it is pi-blocked under either definition. Job J_3 is suspended during $[2, 4)$. It suffers pi-blocking under either definition during $[3, 4)$ since it is among the $c = m = 2$ highest-priority pending jobs. However, J_3 suffers only s-aware pi-blocking during $[2, 3)$ since J_1 is pending but not ready then.

The focus of this paper is locking protocols for s-oblivious schedulability analysis. The rationale for this choice is twofold. First, we are most interested in C-EDF [10, 21], for which G-EDF schedulability tests are required to establish the schedulability of each cluster. As noted above, most G-EDF schedulability tests are s-oblivious. And second, even though s-aware analysis seems intuitively to be much less pessimistic than s-oblivious analysis, locking protocols for s-oblivious analysis can in fact be superior to those for s-aware analysis, as we report in detail in Section 5.

3.2 A Blocking Complexity Measure

As mentioned in Section 2.4, the principal goal in designing a real-time locking protocol is to minimize pi-blocking. Some amount of pi-blocking is inherently unavoidable if (some) resource accesses require mutual exclusion. A locking protocol must hence strike a balance between favoring resource requests of some jobs over those of others. For example, in the extreme, a protocol could guarantee a task to never incur pi-blocking if resources are never granted to other tasks. Clearly, such a protocol is not useful, but it highlights that just considering the pi-blocking bound of only high-priority (or privileged) tasks is not representative of the overall pi-blocking caused by a particular locking protocol.

To compare locking protocols, we thus consider *maximum pi-blocking*, formally $\max_{1 \leq i \leq n}\{b_i\}$, to characterize a protocol's overall blocking behavior. Maximum pi-blocking reflects the per-task bound required for schedulability analysis of the task that incurs the most pi-blocking. It is worth emphasizing that it does *not* necessarily reflect the maximum acquisition delay, which is irrelevant from a schedulability analysis point of view (recall Section 2.4).

Concrete bounds on pi-blocking must necessarily depend on each $L_{i,q}$—long requests will cause long priority inversions under any protocol. Similarly, bounds for any reasonable protocol grow linearly with the maximum number of requests per job.[4] Thus, when deriving asymptotic bounds, we consider, for each T_i, $\sum_{1 \leq q \leq n_r} N_{i,q}$ and each $L_{i,q}$ to be constants and assume $n \geq m$. All other parameters are considered variable (or dependent on m and n). In particular, we do not impose constraints on the ratio $\max\{p_i\}/\min\{p_i\}$, the number of resources n_r, or the number of tasks sharing each ℓ_q. To simplify our notation, we let $L^{max} \triangleq \max\left\{L_{i,q} \mid 1 \leq i \leq n \wedge 1 \leq q \leq n_r\right\}$ denote the the maximum critical section length when deriving asymptotic bounds. To reiterate, we assume $L^{max} = O(1)$.

In accordance with the goal of minimal pi-blocking, we seek to design protocols under which the amount of time lost to pi-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). To this end, we next establish a lower bound on maximum pi-blocking under s-oblivious schedulability analysis.

3.3 Lower Bound on Maximum S-Oblivious Pi-Blocking

The importance of differentiating between s-oblivious and s-aware pi-blocking stems from the fact that each definition gives rise to a different lower bound on maximum pi-blocking. In the case of s-oblivious schedulability analysis, $\Omega(m)$ maximum pi-blocking is unavoidable in some cases. Consider the following pathological high-contention task set.

Definition 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 $n \geq m \geq 2$.

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

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 $\mathcal{R}_{i,1,j}$, where $\mathcal{L}_{i,1,j} = 1$. That is, releases occur in groups of m jobs and each job requires ℓ_1 for its entire computation. A resulting G-EDF schedule is illustrated in Figure 2.

There are n/m groups of m tasks each that release jobs simultaneously. For each group g, where $g \in \{0, \ldots, n/m - 1\}$, jobs of $T_{g \cdot m+1}, \ldots, T_{g \cdot m+m}$ issue 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

[4] Interestingly, in the uniprocessor case, the PCP [43, 47] and the SRP [3] both ensure $O(1)$ maximum pi-blocking regardless of the number of requests, which is possible due to the lack of concurrency (after a job has acquired a resource once, lower-priority jobs cannot lock it again while higher-priority jobs are ready). In the multiprocessor case, resources may be repeatedly locked by concurrently-scheduled remote jobs, which implies that a job may incur pi-blocking each time that it issues a request.

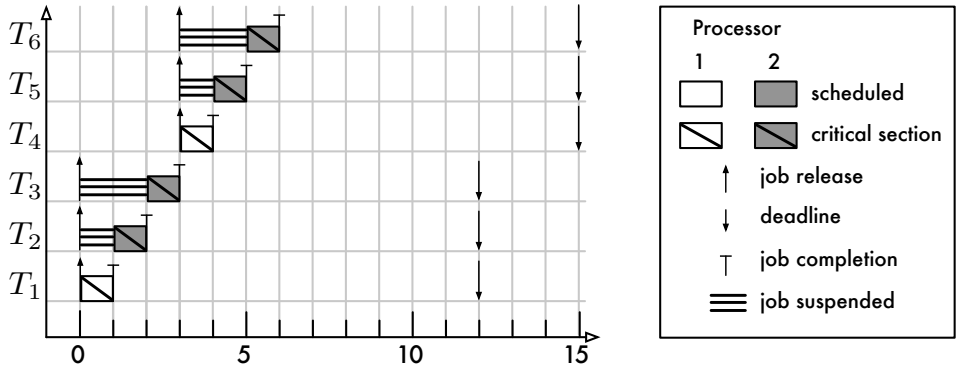


Fig. 2 Illustration of Lemma 1. The depicted example shows a G-EDF schedule of $\tau^{seq}(n)$ for $n = 6$ and $m = 3$, and thus $g \in \{0, 1\}$. The first group of jobs $(J_{1,1}, J_{2,1}, J_{3,1})$ is released at time 0; the second group $(J_{4,1}, J_{5,1}, J_{6,1})$ is released at time 3. Each group incurs $0 + 1 + 2 = \sum_{i=0}^{m-1} i$ total s-oblivious pi-blocking. (Jobs $J_{1,2}$, $J_{2,2}$, and $J_{3,2}$ have been omitted for clarity.)

of pi-blocking for each $d \in \{0, \ldots, m-1\}$. Hence, for each g, $\sum_{i=g \cdot m+1}^{g \cdot m+m} b_i \geq \sum_{i=0}^{m-1} i = \Omega(m^2)$, and thus, across all groups,

$$\sum_{i=1}^{n} b_i = \sum_{g=0}^{(n/m-1)} \sum_{i=g \cdot m+1}^{g \cdot m+m} b_i = \frac{n}{m} \cdot \Omega(m^2) = \Omega(nm),$$

which implies $\max_{1 \leq i \leq n}\{b_i\} = \Omega(m)$.

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 JLFP schedulers as well. The lower bound applies equally to clustered JLFP schedulers with $c < m$ since $\tau^{seq}(n)$ can be trivially partitioned such that each processor serves at least $\lfloor n/c \rfloor$ and no more than $\lceil n/c \rceil$ tasks. □

Perhaps surprisingly, the improvement in analysis accuracy in suspension-aware analysis comes at the cost of an *increased* lower bound for mutex protocols: in prior work, we established a lower bound of $\Omega(n)$ on maximum s-aware pi-blocking [17]. Intuitively, this difference arises because, under s-oblivious analysis, at most m jobs can incur pi-blocking at the same time (a job incurs s-oblivious pi-blocking only if it is among the c highest-priority jobs in its cluster—see Definition 1), whereas no such limit exists for s-aware pi-blocking. That is, under s-oblivious schedulability analysis, high-priority jobs that incur pi-blocking implicitly "shield" lower-priority jobs from incurring pi-blocking at the same time, which allows some of the s-oblivious approach's inherent pessimism to be "reused" to obtain less pessimistic analysis of locking protocols.

The remainder of this paper is exclusively concerned with the design of locking protocols tailored to take full advantage of the properties of s-oblivious schedulability analysis; an in-depth discussion of locking protocols for s-aware pi-blocking and a formal derivation of the $\Omega(n)$ lower bound on maximum s-aware pi-blocking can be found elsewhere [14].

4 The $O(m)$ Locking Protocol Family

In light of the lower bound of $\Omega(m)$ maximum s-oblivous pi-blocking, which is asymptotically tight, an *optimal* s-oblivious locking protocol must ensure $O(m)$ maximum s-oblivious pi-blocking. In fact, an $O(m)$ bound on acquisition delay has long been known to be tight for spin-based protocols: when jobs busy-wait non-preemptively in FIFO order, they must wait for at most $m - 1$ earlier requests (*e.g.*, see [14, 18, 28, 32]). However, prior work has not yielded an $O(m)$ suspension-based locking protocol.

The *family of $O(m)$ locking protocols* (the OMLP family), in part inspired by spin-based protocols and first described in [17, 19], includes a mutex protocol, an RW protocol, and a k-exclusion protocol for clustered scheduling with arbitrary cluster sizes ($1 \leq c \leq m$) and a mutex protocol for the special case of global scheduling.[5] The OMLP family's main features are the following.

- Both mutex protocols ensure maximum pi-blocking that is optimal within a factor that approaches two under s-oblivious analysis. All previously proposed suspension-based locking protocols are asymptotically suboptimal with respect to maximum pi-blocking under s-oblivious analysis.
- The OMLP's RW and k-exclusion variants are the first suspension-based multiprocessor locking protocols of their kind (prior work on suspension-based multiprocessor locking protocols was focused on mutex constraints).
- The RW protocols ensure maximum pi-blocking for writers that is optimal within a factor that approaches four for large m under s-oblivious analysis (the lower bounds on maximum pi-blocking do not apply to readers since they assume mutual exclusion—see Section 4.6 below).
- The k-exclusion protocol ensures $O(\frac{m}{\min\{k_q\}})$ maximum pi-blocking. The ensured bound on maximum pi-blocking is optimal within a factor that approaches two for large m under s-oblivious analysis.
- The OMLP is the first published suspension-based locking protocol that has been designed and analyzed specifically for the case of $1 < c < m$ (prior work focused on global or partitioned scheduling).

The last point, support for truly clustered scheduling, poses significant challenges from a locking perspective because clusters with $1 < c < m$ exhibit aspects of both partitioned and global scheduling, which seem to necessitate fundamentally different means for bounding priority inversions. We begin by describing a novel method for controlling priority inversions that is key to the OMLP's optimality for the case of $1 < c < m$.

4.1 Resource-Holder Progress

To prevent maximum pi-blocking from becoming unbounded or unsuitably large (*i.e.*, bounds should not include job execution costs in addition to request lengths), a locking protocol must ensure that resource-holding jobs progress in their execution when high-priority jobs are waiting. That is, low-priority jobs must be scheduled in spite of their low base priority when they cause other higher-priority jobs to incur pi-blocking. A real-time locking protocol thus requires a mechanism to raise the effective priority of resource holders, either on demand (when a waiting job incurs pi-blocking) or unconditionally. All prior protocols employ priority inheritance or priority boosting to this end—unfortunately, neither generalizes to clustered scheduling with $1 < c < m$.

4.1.1 Limits of Priority Boosting

Priority inheritance is ineffective at bounding maximum priority inversions if $c < m$ because comparing priorities across cluster boundaries is meaningless from an analytical point of view (the highest priority in one cluster may be numerically low in another—an example can be found in [14]). For this reason, all prior protocols for partitioned scheduling instead rely on priority boosting to ensure resource-holder progress. Priority boosting prevents preempted jobs from transitively delaying waiting higher-priority jobs by unconditionally raising the effective priority of resource-holding jobs above that of non-resource-holding jobs. While conceptually simple, the unconditional nature of priority boosting may itself cause pi-blocking. Under partitioning ($c = 1$), this effect can be controlled such that jobs incur at most $O(m)$ s-oblivious pi-blocking [17], but this approach does not extend to $c > 1$. This is best illustrated with an example.

Example 2 For the sake of simplicity, suppose that requests are satisfied in FIFO order, and that a resource holder's priority is boosted. A possible result is shown in Figure 3: jobs of tasks in τ_2 repeatedly request ℓ_1 and ℓ_2 in a pattern that causes low-priority jobs of tasks $T_2, \ldots, T_5$ in τ_1 to be priority-boosted simultaneously.

[5] The initial description of the OMLP [17] contained a variant for partitioned scheduling. This special case is not considered herein because, from an analytical point of view, it has since been superseded by the OMLP's mutex protocol for clustered scheduling.

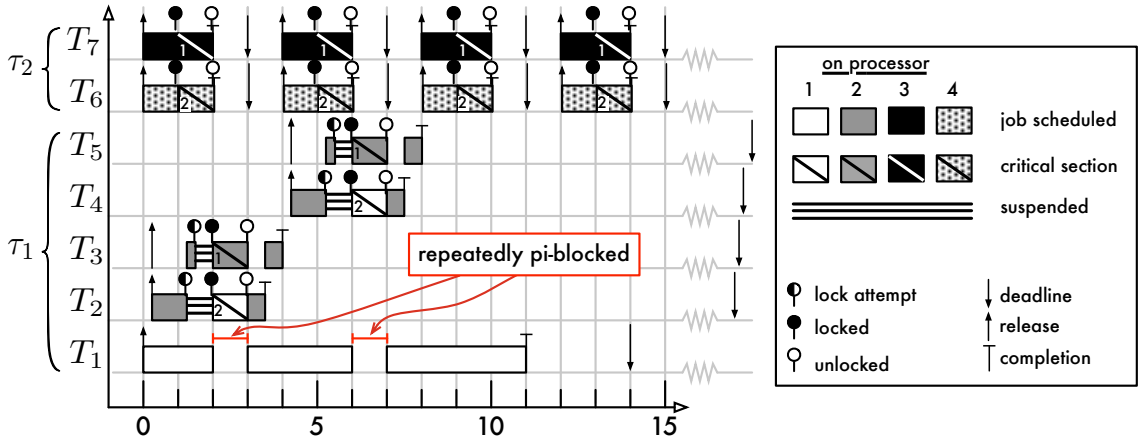


Fig. 3 Seven tasks sharing two resources (ℓ_1, ℓ_2) across two two-processor clusters under C-EDF scheduling (the digit within each critical section indicates which resource was requested)

Whenever $c = 2$ jobs are priority-boosted at the same time, J_1 is necessarily preempted, which causes it to be pi-blocked repeatedly. In general, as c jobs must be priority-boosted to force a preemption, priority boosting may cause $\Omega(\frac{n}{c})$ pi-blocking, which makes it unsuitable for constructing a protocol with $O(m)$ maximum pi-blocking.

The example shows that priority boosting may cause a job to incur pi-blocking repeatedly, and independently of its own requests, if $c > 1$. If instead $c = 1$, then lower-priority jobs cannot issue requests while higher-priority jobs execute and repeated pi-blocking due to priority boosting is not an issue. That is, while priority inheritance fundamentally works only if $m = c$, priority boosting is only appropriate for $c = 1$ and leads to sub-optimal pi-blocking if $c > 1$.

4.1.2 Priority Donation

To overcome the gap in the range of $1 < c < m$, the OMLP uses a novel progress mechanism named *priority donation* that ensures the following two properties.

P1 A resource-holding job is always scheduled.
P2 The duration of s-oblivious pi-blocking caused by the progress mechanism (*i.e.*, the rules that maintain P1) is bounded by the maximum request span (with regard to any job).

Priority boosting unconditionally forces resource holders to be scheduled (Property P1), but it does not specify which job will be preempted as a result. As the example in Figure 3 demonstrates, if $c > 1$, this is problematic since an "unlucky" job can repeatedly be a preemption "victim" (like J_1 in Figure 3), thereby invalidating P2.

Priority donation is a form of priority boosting in which the "victim" is predetermined such that each job is preempted at most once. This is achieved by establishing a *donor relationship* when a potentially harmful job release occurs (*i.e.*, one that could invalidate P1). In contrast to priority boosting, priority donation only takes effect when needed. In the examples and the discussion below, we assume mutex locks for the sake of simplicity; however, the proposed protocol applies equally to RW and k-exclusion locks.

Request rule In the following, let J_i denote a job that requires a resource ℓ_q at time t_1, as illustrated in Figure 4. Priority donation achieves P1 and P2 for $1 \leq c \leq m$ in two steps: it first requires that J_i has a sufficiently high base priority, and then ensures that J_i's effective priority remains high until J_i releases ℓ_q.

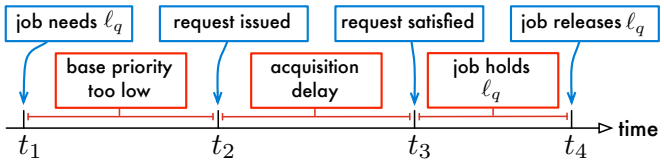


Fig. 4 Request phases under priority donation

D1 J_i may issue a request only if it is among the c highest-priority pending jobs in its cluster (with regard to base priorities). If necessary, J_i suspends until it may issue a request.

Rule D1 ensures that a job has sufficient priority to be scheduled without delay at the time of request. That is, Property P1 holds at time t_2 in Figure 4. However, some—but not all—later job releases during $[t_2, t_4]$ could preempt J_i. Consider a list of all pending jobs in J_i's cluster sorted by decreasing base priority, and let x denote J_i's position in this list at time t_2. In other words, J_i is the x^{th} highest-priority pending job at time t_2. By Rule D1, $x \leq c$. If there are at most $c - x$ higher-priority jobs released during $[t_2, t_4]$, then J_i remains among the c highest-priority pending jobs and no protocol intervention is required. However, when J_i is the c^{th} highest-priority pending job in its cluster, a higher-priority job release may cause J_i to be preempted or to have insufficient priority to be scheduled when it resumes, thereby violating P1. Priority donation intercepts such releases.

Donor rules A *priority donor* is a job that suspends to allow a lower-priority job to complete its request. Each job has at most one priority donor at any time. We first define how jobs become donors and when they suspend, and illustrate the rules with an example thereafter. Let J_d denote J_i's priority donor (if any), and let t_a denote J_d's release time.

D2 J_d becomes J_i's priority donor at time t_a if **(a)** J_i was the c^{th} highest-priority pending job prior to J_d's release (with regard to its cluster), **(b)** J_d has one of the c highest base priorities, and **(c)** J_i has issued a request that is incomplete at time t_a (*i.e.*, $t_a \in [t_2, t_4)$ with regard to J_i's request as illustrated in Figure 4).

D3 J_i inherits the priority of J_d (if any) during $[t_2, t_4)$.

The purpose of Rule D3 is to ensure that J_i will be scheduled if ready. However, J_d's relative priority could decline due to subsequent releases. In this case, the donor role is passed on.

D4 If J_d is displaced from the set of the c highest-priority jobs by the release of J_h, then J_h becomes J_i's priority donor and J_d ceases to be a priority donor. (By Rule D3, J_i thus inherits J_h's priority.)

Rule D4 ensures that J_i remains among the c highest-effective-priority *pending* jobs (with regard to its cluster). The following two rules ensure that J_i and J_d are never ready at the same time, thereby freeing a processor for J_i to be scheduled on.

D5 If J_i is ready when J_d becomes J_i's priority donor (by either Rule D2 or D4), then J_d suspends immediately (*i.e.*, J_d's release is effectively delayed).

D6 If J_d is J_i's priority donor when J_i resumes at time t_3, then J_d suspends (if ready).

Further, a priority donor may not execute a request itself and may not prematurely exit.

D7 A priority donor may not issue requests. J_d suspends if it requires a resource while being a priority donor.

D8 If J_d finishes execution while being a priority donor, then its completion is postponed, that is, J_d suspends and remains pending until it is no longer a priority donor.

J_d may continue once its donation is no longer required, or when a higher-priority job takes over.

D9 J_d ceases to be a priority donor as soon as either **(a)** J_i completes its request (*i.e.*, at time t_4 in Figure 4), **(b)** J_i's base priority becomes one of the c highest (with regard to pending jobs in J_i's cluster), or **(c)** J_d is relieved by Rule D4. If J_d suspended due to Rules D5–D7, then it resumes.

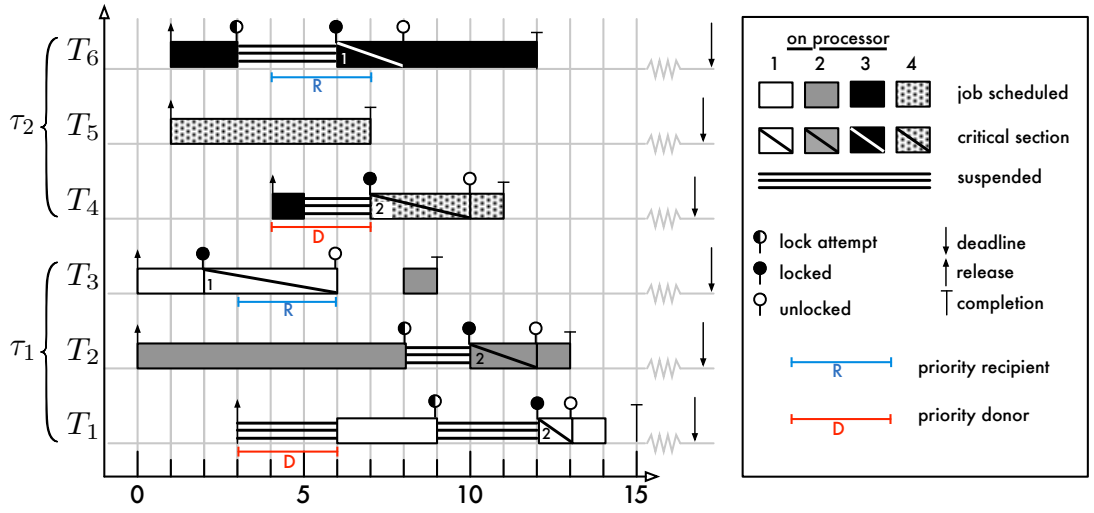


Fig. 5 Six tasks sharing two serially-reusable resources across two two-processor clusters under C-EDF scheduling (the digit within each critical section indicates which resource was requested)

Under a JLFP scheduler, Rule D9b can only be triggered when higher-priority jobs complete.

Example 3 Figure 5 shows a resulting schedule assuming jobs wait in FIFO order. Priority donation occurs first at time 3, when the release of J_1 displaces J_3 from the set of the $c = 2$ highest-priority pending jobs of tasks in τ_1. Since J_3 holds ℓ_1, J_1 becomes J_3's priority donor (Rule D2) and suspends immediately since J_3 is ready (Rule D5). J_1 resumes when its duties cease at time 6 (Rule 9a). If J_1 would not have donated its priority to J_3, then it would have preempted J_3, thereby violating P1. At time 3, J_6 also requests ℓ_1 and suspends as ℓ_1 is unavailable. It becomes a priority recipient when J_4 is released at time 4 (Rule D2). Since J_6 is already suspended, Rule D5 does not apply and J_4 remains ready. However, at time 5, J_4 requires ℓ_2, but since it is still a priority donor, it may not issue a request and must suspend instead (Rule D7). J_4 may resume and issue its request at time 7 since J_5 finishes, which causes J_6 to become one of the two highest-priority pending jobs of tasks in τ_2 (Rule 9b). If priority donors were allowed to issue requests, then J_4 would have been suspended while holding ℓ_2 when J_6 resumed at time 6, thereby violating P1.

Taken together, Rules D1–D9 ensure resource-holder progress under clustered scheduling with arbitrary cluster sizes ($1 \leq c \leq m$).

Lemma 2 *Priority donation ensures Property P1.*

Proof Rule D7 prevents Rules D5 and D6 from suspending a resource-holding job. Rule D1 establishes Property P1 at time t_2. If J_i's base priority becomes insufficient to guarantee P1, its effective priority is raised by Rules D2 and D3. Rules D4 and D8 ensure that the donated priority is always among the c highest (with regard to pending jobs in J_i's cluster), which, together with Rules D5 and D6, effectively reserves a processor for J_i to run on when ready. □

By establishing the donor relationship at release time, priority donation ensures that a job is a "preemption victim" at most once, even if $c > 1$.

Lemma 3 *Priority donation ensures Property P2.*

Proof A job incurs s-oblivious pi-blocking if it is among the c highest-priority pending jobs in its cluster and either **(i)** suspended or **(ii)** ready and not scheduled (*i.e.*, preempted). We show that (i) is bounded and that (ii) is impossible.

Case (*i*). Only Rules D1 and D5–D8 cause a job to suspend. Rule D1 does not cause s-oblivious pi-blocking: the interval $[t_1, t_2)$ ends as soon as J_i becomes one of the c highest-priority pending jobs. Rules D5–D8 apply to priority donors. J_d becomes a priority donor only immediately upon release or not at all (Rules D2 and D4), that is, each J_d donates its priority to some J_i only once. By Rule D2, the donor relationship starts no earlier than t_2, and, by Rule D9, ends at the latest at time t_4. By Rules D8 and D9, J_d either resumes or completes when it ceases to be a priority donor. J_d suspends thus for at most the duration of one entire request span.

Case (*ii*). Let J_x denote a job that is ready and among the c highest-priority pending jobs (with regard to base priorities) of tasks in cluster τ_j, but not scheduled. Let A denote the set of ready jobs of tasks in τ_j with higher base priorities than J_x, and let B denote the set of ready jobs of tasks of τ_j with higher effective priorities than J_x that are not in A. Only jobs in A and B can preempt J_x. Let D denote the set of priority donors of jobs in B.

By Rule D3, every job in B has a priority donor that is, by construction, unique: $|B| = |D|$. By assumption, $|A| + |B| \geq c$ (otherwise J_x would be scheduled), and thus also $|A| + |D| \geq c$.

Rules D5 and D6 imply that no job in D is ready (since every job in B is ready): $A \cap D = \emptyset$, and hence $|A \cup D| = |A| + |D|$.

By the definition of B, every job in D has a base priority that exceeds J_x's base priority. Similarly, by the definition of A, every job in A has a higher base priority than J_x as well. Thus, since every job in $A \cup D$ has a higher base priority than J_x, there exist $|A \cup D| = |A| + |D| \geq c$ pending jobs of tasks in τ_j with higher base priority than J_x. Contradiction. □

Priority donation further limits maximum concurrency, which is key to the analysis of the protocols presented next in Sections 4.2–4.4.

Lemma 4 *Let $R_j(t)$ denote the number of requests issued by jobs of tasks in cluster τ_j that are incomplete at time t. Under priority donation, $R_j(t) \leq c$ at all times.*

Proof Similar to Case (ii) above. Suppose $R_j(t) > c$ at time t. Let H denote the set of the c highest-priority pending jobs of tasks in τ_j (at time t and with regard to base priorities), and let I denote the set of jobs of tasks in τ_j that have issued a request that is incomplete at time t.

Let A denote the set of high-priority jobs with incomplete requests (*i.e.*, $A = H \cap I$) and let B denote the set of low-priority jobs with incomplete requests (*i.e.*, $B = I \setminus A$).

Let D denote the set of priority donors of jobs in B. Together, Rules D2, D4, D8, and D9 ensure that every job in B has a unique priority donor. Therefore $|B| = |D|$.

By definition, $|A| + |B| = |I| = R_j(t)$. By our initial assumption, this implies $|A| + |B| > c$ and thus $|A| + |D| > c$. By Rules D2 and D4, $D \subseteq H$ (only high-priority jobs are donors).

By Rule D7, $A \cap D = \emptyset$ (donors may not issue requests). Since, by definition, $A \subseteq H$, this implies $|H| \geq |A| + |D| > c$. Contradiction. □

In the following, we show that Lemmas 2–4 provide a strong foundation that enables the design of simple, yet asymptotically optimal, locking protocols.

4.2 The Clustered OMLP for Mutual Exclusion

We begin with ensuring mutex constraints for $1 \leq c \leq m$, which is the most straightforward case. An asymptotically optimal mutex protocol can be layered on top of priority donation by using simple FIFO queues just as they are used in non-preemptive spinlocks. The following protocol's simplicity demonstrates that priority donation is a powerful aid for worst-case analysis.

Structure For each serially-reusable resource ℓ_q, there is a FIFO queue FQ_q that is used to serialize conflicting accesses. The job at the head of FQ_q holds ℓ_q.

Rules Access to each resource is granted according to the following rules. Let J_i denote a job that issues a request $\mathcal{R}_{i,q,v}$ for ℓ_q.

X1 J_i is enqueued in FQ_q when it issues $\mathcal{R}_{i,q,v}$. If FQ_q was non-empty, then J_i suspends until $\mathcal{R}_{i,q,v}$ is satisfied.
X2 $\mathcal{R}_{i,q,v}$ is satisfied when J_i becomes the head of FQ_q.
X3 J_i is dequeued from FQ_q when $\mathcal{R}_{i,q,v}$ is complete. The new head of FQ_q (if any) is resumed.

Rules X1–X3 correspond to times t_2–t_4 in Figure 4.

Example 4 Figure 5 depicts an example of the clustered OMLP for serially-reusable resources. (Figure 5 was previously discussed in the context of priority donation.) At time 2, J_3 requests ℓ_1 and is enqueued in FQ_1 (Rule X1). Since FQ_1 was empty, J_3's request is satisfied immediately (Rule X2). When J_6 requests the same resource at time 3, it is appended to FQ_1 and suspends. When J_3 releases ℓ_1 at time 6, J_6 becomes the new head of FQ_1 and resumes (Rule X3). At time 7, J_4 acquires ℓ_2 and enqueues in FQ_2, which causes J_2 and J_1 to suspend when they, too, request ℓ_2 at times 8 and 9. Importantly, priorities are ignored in each FQ_q: when J_4 releases ℓ_2 at time 10, J_2 becomes the resource holder and is resumed, even though J_1 has a higher base priority. While using FIFO queues instead of priority queues in real-time systems may seem counterintuitive, priority queues are in fact problematic in a multiprocessor context since they allow starvation, which renders them unsuitable for constructing protocols with $O(m)$ maximum pi-blocking (as discussed in more detail in Section 4.5 below).

Priority donation is in two ways crucial to the OMLP: requests complete without delay and maximum contention is limited.

Lemma 5 *At most m jobs are enqueued in any FQ_q.*

Proof By Lemma 4, at most c requests are incomplete at any point in time in each cluster. Since there are $\frac{m}{c}$ clusters, no more than $\frac{m}{c} \cdot c = m$ jobs are enqueued in any FQ_q. □

Lemma 6 *A job J_i that requests a resource ℓ_q incurs acquisition delay for the duration of at most $m - 1$ requests.*

Proof By Lemma 5, at most $m - 1$ other jobs precede J_i in FQ_q. By Lemma 2, the job at the head of FQ_q is scheduled. Therefore, J_i becomes the head of FQ_q after the combined length of at most $m - 1$ requests. □

This property suffices to prove asymptotic optimality.

Theorem 1 *The clustered OMLP for serially-reusable resources causes a job J_i to incur at most $b_i = m \cdot L^{max} + \sum_{q=1}^{n_r} N_{i,q} \cdot (m-1) \cdot L^{max} = O(m)$ s-oblivious pi-blocking.*

Proof By Lemma 3, the duration of s-oblivious pi-blocking caused by priority donation is bounded by the maximum request span. Recall from Section 2.3 that the request span includes both the request length and any acquisition delay. By Lemma 6, maximum acquisition delay per request is bounded by $(m-1) \cdot L^{max}$. The maximum request span is thus bounded by $m \cdot L^{max}$. Recall from Section 3.2 that $\sum_{q=1}^{n_r} N_{i,q}$ and L^{max} are presumed constant. The bound follows. □

The protocol for serially-reusable resources is thus asymptotically optimal with regard to maximum s-oblivious pi-blocking. A practical, non-asymptotic bound on maximum pi-blocking that takes individual request lengths and frequencies into account is derived in Appendix A.

4.3 The Clustered OMLP for RW Exclusion

In throughput-oriented computing, RW locks are attractive because they increase average concurrency (compared to mutex locks) if read requests are more frequent than write requests. In a real-time context, RW locks should also lower pi-blocking for readers, that is, the higher degree of concurrency must be reflected in *a priori* worst-case analysis and not just in observed average-case delays.

Unfortunately, many RW lock types commonly in use in throughput-oriented systems provide only little analytical benefits because they either allow starvation or serialize readers [18]. As an example for the former, consider *reader preference* RW locks, under which write requests are only satisfied if there are no unsatisfied read requests. Such locks have the advantage that a read request incurs only $O(1)$ acquisition delay, but they also expose write requests to potentially unbounded acquisition delays. In contrast, *task-fair* RW locks, in which requests (either read or write) are satisfied strictly in FIFO order, are an example for the latter case: in the worst case, read requests and write requests are interleaved such that read requests incur $\Omega(m)$ acquisition delay (assuming priority donation), just as they would under a mutex lock.

In recent work on spin-based RW locking protocols [18], we introduced *phase-fair* RW locks as an alternative better suited to reducing worst-case delays. Phase-fairness is defined by the following key properties.

- Reader phases and writer phases alternate (unless there are only requests of one kind).
- At the beginning of a reader phase, all incomplete read requests are satisfied.
- One write request is satisfied at the beginning of a writer phase.
- Read requests are allowed to join a reader phase that is already in progress only if there are no incomplete write requests.

This results in $O(1)$ acquisition delay for read requests without starving write requests [14, 18]. Note that this does not contradict the lower bound on s-oblivious pi-blocking (Lemma 1) because the lower bound depends on mutual exclusion. It thus only applies to write requests (which must be exclusive), but not to read requests (which may be satisfied concurrently with other read requests).

The following rules realize a suspension-based phase-fair RW lock.

Structure For each RW resource ℓ_q, there are three queues: a FIFO queue for writers, denoted WQ_q, and two reader queues RQ_q^1 and RQ_q^2. Initially, RQ_q^1 is the *collecting* and RQ_q^2 is the *draining* reader queue. The roles, denoted as CQ_q and DQ_q, switch as reader and writer phases alternate; that is, the designations "collecting" and "draining" are not static.

Reader rules Let J_i denote a job that issues a read request $\mathcal{R}_{i,q,v}^R$ for ℓ_q. The distinction between CQ_q and DQ_q serves to separate reader phases. Readers always enqueue in the (at the time of request) collecting queue. If queue roles change, then a writer phase starts when the last reader releases ℓ_q.

R1 J_i is enqueued in CQ_q when it issues $\mathcal{R}_{i,q,v}^R$. If WQ_q is non-empty (*i.e.*, if there are one or more writers present), then J_i suspends.

R2 $\mathcal{R}_{i,q,v}^R$ is satisfied either immediately if WQ_q is empty when $\mathcal{R}_{i,q,v}^R$ is issued, or when J_i is subsequently resumed (by an exiting writer, see Rule W3 below).

R3 Let RQ_q^y denote the reader queue in which J_i was enqueued due to Rule R1. J_i is dequeued from RQ_q^y when $\mathcal{R}_{i,q,v}^R$ is complete. If RQ_q^y is DQ_q and J_i is the last job to be dequeued from RQ_q^y, then the current reader phase ends and the head of WQ_q is resumed (WQ_q is non-empty in this case because queue roles changed).

Writer rules Let J_w denote a job that issues a write request $\mathcal{R}_{w,q,v}^W$ for ℓ_q. Conflicting writers wait in FIFO order. The writer at the head of WQ_q is further responsible for starting and ending reader phases by switching the reader queues.

W1 J_w is enqueued in WQ_q when it issues $\mathcal{R}_{w,q,v}^W$. J_w suspends until $\mathcal{R}_{w,q,v}^W$ is satisfied, unless $\mathcal{R}_{w,q,v}^W$ is satisfied immediately. If WQ_q is empty and CQ_q is not, then the roles of CQ_q and DQ_q are switched to end the current reader phase.

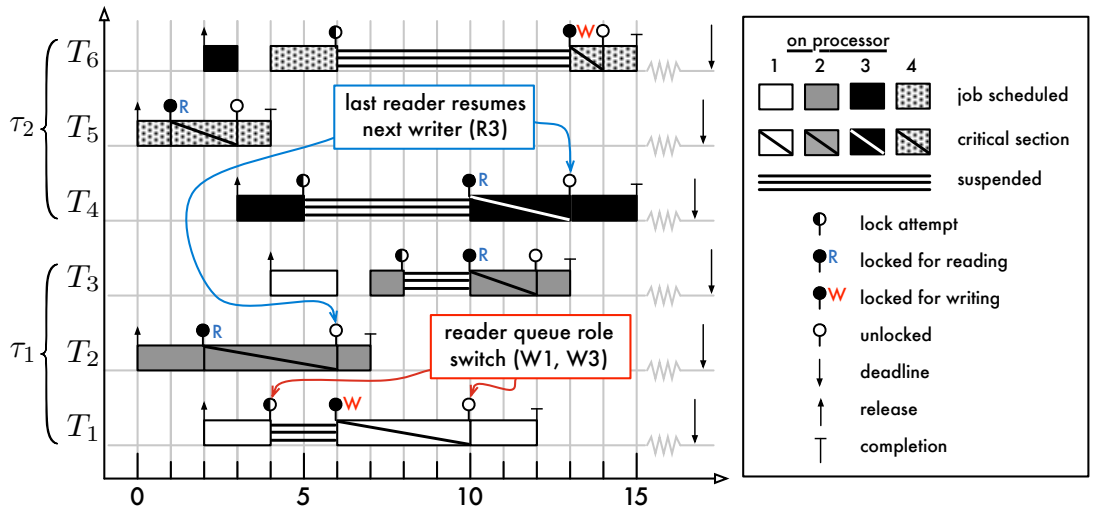


Fig. 6 Six tasks sharing one RW resource across two two-processor clusters under C-EDF scheduling (priority donation does not occur in this example schedule)

W2 $\mathcal{R}^W_{w,q,v}$ is satisfied either immediately if WQ_q and CQ_q are both empty when $\mathcal{R}^W_{w,q,v}$ is issued,[6] or when J_w is subsequently resumed.

W3 J_w is dequeued from WQ_q when $\mathcal{R}^W_{w,q,v}$ is complete. If CQ_q is empty, then the new head of WQ_q (if any) is resumed. Otherwise, each job in CQ_q is resumed and, if WQ_q remains non-empty (*i.e.*, if there are waiting writers), the roles of CQ_q and DQ_q are switched.

Rules R1–R3 and W1–W3 correspond to times t_2–t_4 in Figure 4 (respectively), and are illustrated in Figure 6.

Example 5 Figure 6 depicts six tasks in two clusters sharing one resource. The resource ℓ_1 is first read by J_5, which is enqueued in RQ^1_q, the initial collecting queue, at time 1 (Rule R1). When J_2 issues a read request at time 1, it is also enqueued and its request is satisfied immediately since WQ_1 is still empty (Rule R2). J_1 issues a write request at time 4. Since CQ_1 is non-empty, the roles of CQ_1 and DQ_1 are switched, that is, RQ^1_q becomes the draining reader queue, and J_1 suspends (Rule W1). J_4 issues a read request soon thereafter and is enqueued in RQ^2_q (Rule R1), which is the collecting queue after the role switch. J_4 suspends since WQ_1 is not empty (Rule R2), even though J_2 is still executing a read request. This is required to ensure that write requests are not starved. The reader phase ends when J_2 releases ℓ_1 at time 6, and the next writer, J_1, is resumed (Rules R3 and W2). J_1 releases ℓ_1 and resumes all readers that have accumulated in RQ^2_q (J_3 and J_4). Since WQ_1 is non-empty (J_6 was enqueued at time 6), RQ^2_q becomes the draining reader queue (Rule W3). Under task-fair RW locks, J_3 would have remained suspended since it requested ℓ_1 after J_6. In contrast, J_6 must wait until the next writer phase at time 13 and *all* waiting readers are resumed at the beginning of the next reader phase at time 10 (Rule W3).

Together with priority donation, the reader and writer rules above realize a phase-fair RW lock. Due to the intertwined nature of reader and writer phases, we first consider the head of WQ_q (a writer phase), then CQ_q (a reader phase), and finally the rest of WQ_q.

Lemma 7 *Let J_w denote the head of WQ_q. J_w incurs acquisition delay for the duration of at most one read request length before its request is satisfied.*

[6] If WQ_q and CQ_q are both empty, then DQ_q is necessarily empty, too, as any readers in the draining queue would have had to enqueue when it was still the collecting queue (Rule R1) and the roles of CQ_q and DQ_q are only switched when a writer is waiting (Rules W1 and W3).

Proof J_w became head of WQ_q in one of two ways: by Rule W1 (if WQ_q was empty prior to J_w's request) or by Rule W3 (if J_w had a predecessor in WQ_q). In either case, there was a reader queue role switch when J_w became head of WQ_q (unless there were no unsatisfied read requests, in which case the claim is trivially true). By Rule R3, if a reader phase delayed J_w, then J_w is resumed as soon as the last reader in DQ_q releases ℓ_q. By Rule R1, no new readers enter DQ_q. Due to priority donation, there are at most $m-1$ jobs in DQ_q (Lemma 4), and each job holding ℓ_q is scheduled (Lemma 2). The claim follows. □

Lemma 8 *Let J_i denote a job that issues a read request for ℓ_q. J_i incurs acquisition delay for the combined duration of at most one read and one write request.*

Proof If WQ_q is empty, then J_i's request is satisfied immediately (Rule R2). Otherwise, it suspends and is enqueued in CQ_q (Rule R1). This prevents consecutive writer phases (Rule W3). J_i's request is thus satisfied as soon as the current head of WQ_q releases ℓ_q (Rule W3). By Lemma 7, the head of WQ_q incurs acquisition delay for no more than the length of one read request (which transitively impacts J_i). Due to priority donation, the head of WQ_q is scheduled when its request is satisfied (Lemma 2). Therefore, J_i waits for the duration of at most one read and one write request. □

Lemma 8 shows that readers incur $O(1)$ acquisition delay. Next, we show that writers incur $O(m)$ acquisition delay.

Lemma 9 *Let J_w denote a job that issues a write request for ℓ_q. J_w incurs acquisition delay for the duration of at most $m-1$ write and m read requests before its request is satisfied.*

Proof It follows from Lemma 4 that at most $m-1$ other jobs precede J_w in WQ_q (analogously to Lemma 5). By Lemma 2, J_w's predecessors together hold ℓ_q for the duration of at most $m-1$ write requests. By Lemma 7, each predecessor incurs acquisition delay for the duration of at most one read request once it has become the head of WQ_q. Thus, J_w incurs transitive acquisition delay for the duration of at most $m-1$ read requests before it becomes head of WQ_q, for a total of at most $m-1+1=m$ read requests. □

These properties suffice to prove asymptotic optimality with regard to maximum s-oblivious pi-blocking.

Theorem 2 *The clustered* OMLP *for RW resources causes a job J_i to incur at most*

$$b_i = 2 \cdot m \cdot L^{max} + \left(\sum_{q=1}^{n_r} N_{i,q}^R \cdot 2 \cdot L^{max} \right) + \left(\sum_{q=1}^{n_r} N_{i,q}^W \cdot (2 \cdot m - 1) \cdot L^{max} \right) = O(m)$$

s-oblivious pi-blocking.

Proof By Lemma 3, the duration of s-oblivious pi-blocking caused by priority donation is bounded by the maximum request span. By Lemma 9, maximum acquisition delay per write request is bounded by $(2m-1) \cdot L^{max}$; by Lemma 8, maximum acquisition delay per read request is bounded by $2 \cdot L^{max}$. The maximum request span is thus bounded by $2 \cdot m \cdot L^{max}$. Recall from Section 3.2 that L^{max} and $\sum_{q=1}^{n_r} N_{i,q}$, and hence also $\sum_{q=1}^{n_r} N_{i,q}^W$ and $\sum_{q=1}^{n_r} N_{i,q}^R$, are constants. The bound follows. □

A detailed, non-asymptotic bound on maximum pi-blocking that takes individual request lengths and frequencies into account is given in Appendix A. While the presented analysis assumes phase-fairness, other RW request orders such as task-fairness or preference locks could similarly be implemented on top of priority donation; see [14, 18] for appropriate analysis of task-fair and preference RW locks.

4.4 The Clustered OMLP for k-Exclusion

For some resource types, one option to reduce contention is to *replicate* them. For example, if potential overload of a DSP co-processor is found to pose a risk in the design phase, the system designer could introduce additional instances to improve response times.

As with multiprocessors, there are two fundamental ways to allocate replicated resources: either each task may only request a specific replica, or every task may request any replica. The former approach, which corresponds to partitioned scheduling, has the advantage that a mutex protocol suffices, but it also implies that some resource replicas may idle while jobs wait to acquire their designated replica. The latter approach, equivalent to global scheduling, avoids such bottlenecks, but needs a k-exclusion protocol to do so. Priority donation yields such a protocol for clustered scheduling.

Recall that k_q denotes the number of replicas of resource ℓ_q. In the following, we assume $1 \leq k_q \leq m$. The case of $k_q > m$ is discussed in Section 4.6 below.

Structure Jobs waiting for a replicated resource ℓ_q are kept in a FIFO queue denoted as KQ_q. The replica set RS_q contains all idle instances of ℓ_q. If $\text{RS}_q \neq \emptyset$, then KQ_q is empty.

Rules Let J_i denote a job that issues a request $\mathcal{R}_{i,q,v}$ for ℓ_q.

K1 If $\text{RS}_q \neq \emptyset$, then J_i acquires an idle replica from RS_q. Otherwise, J_i is enqueued in KQ_q and suspends.
K2 $\mathcal{R}_{i,q,v}$ is satisfied either immediately (if $\text{RS}_q \neq \emptyset$ at the time of request) or when J_i is removed from KQ_q.
K3 If KQ_q is non-empty when $\mathcal{R}_{i,q,v}$ completes, the head of KQ_q is dequeued, resumed, and acquires J_i's replica. Otherwise, J_i's replica is released into RS_q.

As it was the case with the definition of the previous protocols, Rules K1–K3 correspond to times t_2–t_4 in Figure 4.

Example 6 Figure 7 depicts an example schedule for one resource (ℓ_1) with $k_1 = 2$. J_5 obtains a replica from RS_1 at time 2 (Rule K1). The second replica of ℓ_1 is acquired by J_2 at time 4. As RS_1 is now empty, J_1 is enqueued in KQ_1 and suspends when it requests ℓ_1 at time 5. However, it is soon resumed when J_5 releases its replica at time 6 (Rule K3). This illustrates one advantage of using k-exclusion locks: if instead one replica would have been statically assigned to each cluster (which reduces the resource-sharing problem to a mutex constraint), then J_1 would have continued to wait while τ_2's replica would have idled. This happens again at time 12: since no job of tasks in τ_1 requires ℓ_1 at the time, both instances are used by jobs of tasks in τ_2.

As with the previous protocols, priority donation is essential to ensure progress and to limit contention.

Lemma 10 *At most $m - k_q$ jobs are enqueued in KQ_q.*

Proof Lemma 4 implies that there are at most m incomplete requests. Since only jobs waiting for ℓ_q are enqueued in KQ_q, at most $m - k_q$ jobs are enqueued in KQ_q. □

Lemma 11 *Let J_i denote a job that issues a request $\mathcal{R}_{i,q,v}$ for ℓ_q. J_i incurs acquisition delay for the duration of at most $\lceil (m - k_q)/k_q \rceil$ maximum request lengths.*

Proof By Lemma 10, at most $m - k_q$ requests must complete before J_i's request is satisfied ($m - k_q - 1$ for J_i to become the head of KQ_q, and one more for J_i to be dequeued). Rules K1 and K3 ensure that all replicas are in use whenever jobs wait in KQ_q. Since resource holders are always scheduled due to priority donation (Lemma 2), requests are satisfied at a rate of at least k_q requests per maximum request length until $\mathcal{R}_{i,q,v}$ is satisfied. The stated bound follows. □

Lemma 11 shows that J_i incurs at most $O(\frac{m}{k_q})$ pi-blocking per request (and none if $k_q = m$). This suffices to show asymptotic optimality.

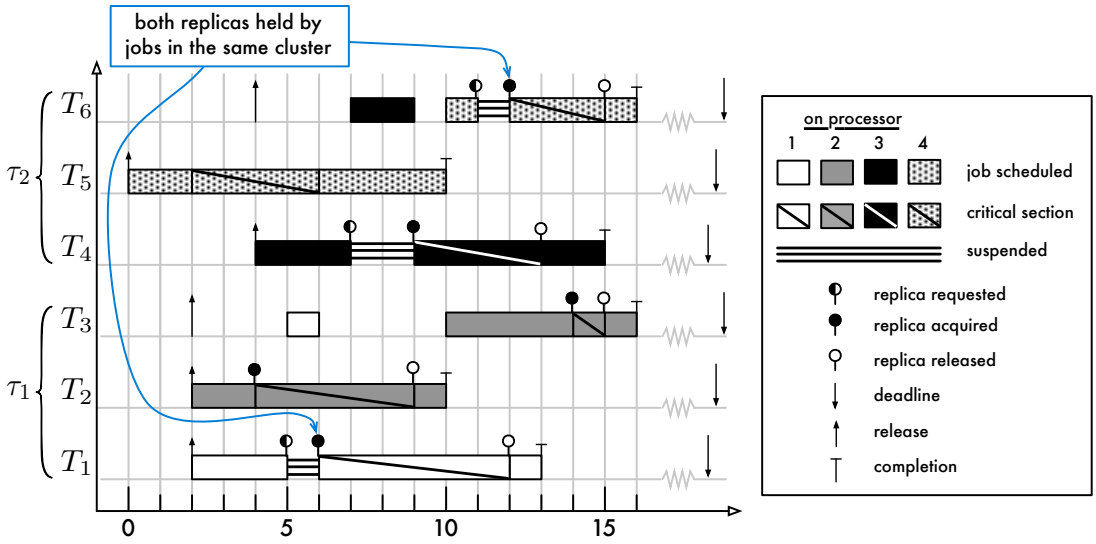


Fig. 7 Six tasks sharing two instances of one resource across two two-processor clusters under C-EDF scheduling (priority donation does not occur in this particular example)

Definition 4 Let $k^{min} \triangleq \min_{1 \leq q \leq r}\{k_q\}$ denote the minimum degree of replication.

Theorem 3 *The clustered OMLP for replicated resources causes a job J_i to incur at most*

$$\begin{aligned} b_i &= \left(1 + \left\lceil \frac{m - k^{min}}{k^{min}} \right\rceil\right) \cdot L^{max} + \sum_{q=1}^{n_r} \left(N_{i,q} \cdot \left\lceil \frac{m - k_q}{k_q} \right\rceil\right) \cdot L^{max} \\ &\leq \left(1 + \left\lceil \frac{m - k^{min}}{k^{min}} \right\rceil\right) \cdot L^{max} + \sum_{q=1}^{n_r} \left(N_{i,q} \cdot \left\lceil \frac{m - k^{min}}{k^{min}} \right\rceil\right) \cdot L^{max} \\ &= O(m/k^{min}) \end{aligned}$$

s-oblivious pi-blocking.

Proof By Lemma 11, maximum acquisition delay per request for ℓ_q is bounded by $\lceil (m - k_q)/k_q) \rceil \cdot L^{max}$. The maximum request span is thus bounded by $(\lceil (m - k^{min})/k^{min}) \rceil + 1) \cdot L^{max}$. Lemma 3 limits the duration of s-oblivious pi-blocking due to priority donation to the maximum request span. The bound follows since $\sum_{q=1}^{n_r} N_{i,q}$ and L^{max} are constants (Section 3.2). □

A detailed, non-asymptotic bound on maximum pi-blocking that takes individual request lengths and frequencies into account is provided in Appendix A. Theorem 3 implies asymptotic optimality for any $k^{min} \leq m$. While Lemma 1 applies only to mutual exclusion (*i.e.*, $k^{min} = 1$), it is trivial to extend the argument to $1 \leq k^{min} \leq m$.

Lemma 12 *There exists an arrival sequence for $\tau^{seq}(n)$ such that, under s-oblivious analysis, $\max_{1 \leq i \leq n}\{b_i\} = \Omega(m/k^{min})$ under any k-exclusion locking protocol and JLFP scheduler, where $1 \leq k^{min} < m$.*

Proof Analogously to Lemma 1. Recall from Definition 3 that $\tau^{seq}(n)$ consists of n tasks, and that each job of each task requires a shared resource ℓ_1 for the entirety of its computation (*i.e.*, $e_i = L_{i,1} = L^{max} = 1$). If there are $k_1 = k^{min}$ replicas of ℓ_1, then at most k^{min} jobs are scheduled at any time. As in the proof of Lemma 1, consider the arrival sequence shown in Figure 2: if m jobs request ℓ_1 simultaneously, then any k-exclusion

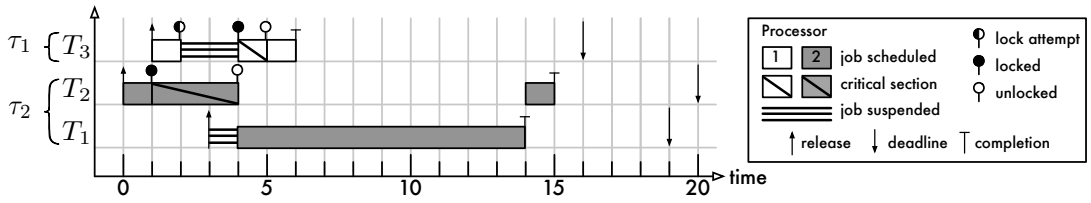


Fig. 8 Pi-blocking of independent jobs under the clustered OMLP and P-EDF scheduling on $m = 2$ processors with $c = 1$

protocol must impart an order among the requests such that only k^{min} requests are satisfied concurrently. To complete each of the m concurrent requests, the k^{min} replicas must be used for $m \cdot L^{max}$ time units in total. This implies that the last request to be satisfied completes no earlier than $m \cdot L^{max}/k^{min}$ time units after it was issued. Therefore, it incurred at least

$$\frac{m \cdot L^{max}}{k^{min}} - L^{max} = \left(\frac{m}{k^{min}} - 1\right) \cdot L^{max}$$

acquisition delay. Further, as each request is sequential and since all requests are of uniform length $L^{max} = 1$, requests are only satisfied at times that are integer multiples of L^{max} (*i.e.*, requests are satisfied only $x \cdot L^{max}$ time units after they are issued, where $0 \leq x \leq \lceil m/k^{min} \rceil - 1$). Therefore, the last of the m concurrent requests to complete was not satisfied until

$$\left\lceil \frac{m}{k^{min}} - 1 \right\rceil \cdot L^{max} = \Omega\left(\frac{m}{k^{min}}\right)$$

time units after the requests were issued. Since at most m jobs are pending at any time in the periodic arrival sequence shown in Figure 2, this implies that $\Omega(m/k^{min})$ s-oblivious pi-blocking is unavoidable in the general case. □

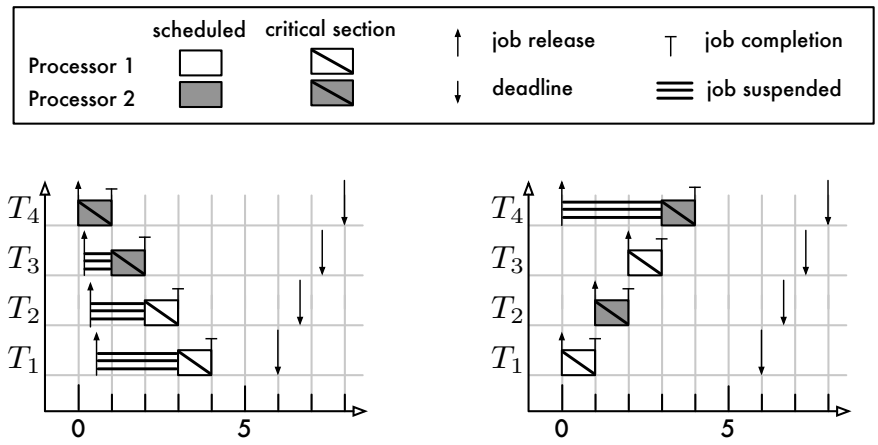
The clustered OMLP for replicated resources is hence asymptotically optimal with regard to maximum s-oblivious pi-blocking.

4.5 An Independence-Preserving Mutex Protocol

As demonstrated in the preceding sections, the primary advantage of priority donation is that it enables simple, asymptotically optimal locking protocols. An undesirable property of priority donation is that *every* task is subject to potential pi-blocking—even those that are independent—because any job may be required to serve as a priority donor upon release. While undesirable, this is fundamental to lock-based real-time synchronization if $c < m$, that is, if priority inversions must be bounded, there is more than one cluster, and tasks may not migrate across cluster boundaries. This is illustrated in Figure 8.

Example 7 Even though job J_1 is independent, it incurs pi-blocking when it serves as J_2's priority donor during $[3, 4)$. This example demonstrates that, if jobs may not migrate across cluster boundaries, it is in general unavoidable for independent jobs to be subject to pi-blocking: if J_1 were allowed to preempt J_2 (to avoid being pi-blocked), then J_3 would incur pi-blocking for the entire duration of J_1's execution (*i.e.*, J_3 would incur a potentially unbounded priority inversion).

Luckily, in the special case of global scheduling (*i.e.*, if $c = m$), it is possible to design locking protocols based on priority inheritance under which independent jobs never incur s-oblivious pi-blocking. That is, in the following, we seek to design an "independence-preserving" locking protocol under which jobs incur pi-blocking only due to resources on which they depend.



(a) Requests satisfied in FIFO order. (b) Requests satisfied in order of job priority.

Fig. 9 G-EDF schedules of $n = 4$ tasks sharing one resource ℓ_1 on $m = 2$ processors

Definition 5 Let $b_{i,q}$ denote an upper bound on maximum pi-blocking incurred by J_i due to requests by any job of any task for resource ℓ_q. A locking protocol is *independence-preserving* if and only if $N_{i,q} = 0 \Rightarrow b_{i,q} = 0$.

Importantly, a task that does not require any shared resources does not incur any pi-blocking under an independence-preserving locking protocol. In this section, we present such a protocol, namely the *global* OMLP for mutex constraints.

4.5.1 Wait Queue Choices

The OMLP variant for clustered scheduling relies on simple FIFO queues to serialize conflicting resource requests. Unfortunately, when FIFO queues are combined with priority inheritance (which, unlike priority donation, does not limit the maximum number of incomplete requests), jobs can incur $\Omega(n)$ s-oblivious pi-blocking. As demonstrated in Figure 9(a), the job with the highest priority (J_1) may incur $\Omega(n)$ pi-blocking if its request is issued just after all other requests.

As priority inheritance is used together with priority queues in the uniprocessor case, (*e.g.*, in the PIP and PCP), it is perhaps not surprising that FIFO ordering by itself is ill-suited to ensuring $O(m)$ maximum pi-blocking. However, ordering requests by job priority, as for instance done in the PPCP [29], 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. This is illustrated in Figure 9 (b), which shows that a job's request may be deferred repeatedly even though it is among the m highest-priority jobs. Thus, ordering *all* requests by job priority is, at least asymptotically speaking, not preferable to the simpler FIFO queuing, and can in fact give rise to $\Omega(mn)$ pi-blocking if tasks with short periods create intense contention [14, 17].

4.5.2 The Global OMLP for Mutual Exclusion

Fortunately, it is possible to use priority inheritance to realize $O(m)$ maximum s-oblivious pi-blocking by combining FIFO and priority ordering. In the global OMLP, each resource is protected by two locks: a priority-based m-exclusion lock that limits access to a regular FIFO mutex lock, which in turn serializes access to the resource. This idea is formalized by the following rules.

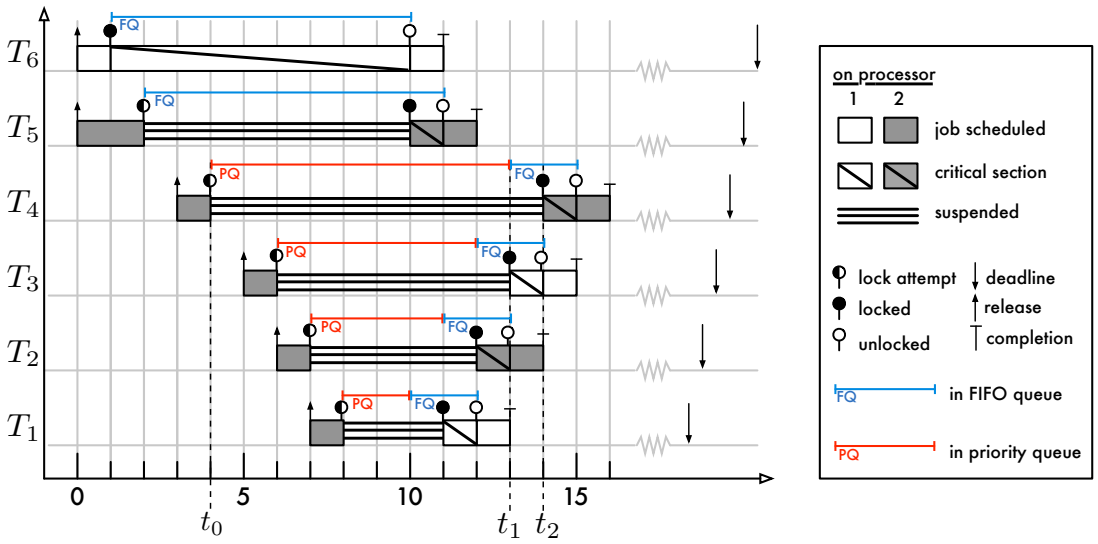


Fig. 10 OMLP mutex protocol for global scheduling under G-EDF for six tasks sharing one resource on $m = 2$ processors

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

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

G1 A job J_i that issues a request $\mathcal{R}_{i,q,v}$ for ℓ_q at time t is appended to FQ_q if $queued_q(t) < m$; otherwise, if $queued_q(t) \geq m$, it is added to PQ_q. $\mathcal{R}_{i,q,v}$ is satisfied when J_i becomes the head of FQ_q.

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

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

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 shown in Figure 10.

Example 8 Figure 10 depicts a G-EDF schedule of six jobs sharing one resource ℓ_1 on $m = 2$ processors under the OMLP's global mutex protocol. At time 1, J_6 requests ℓ_1 and enters FQ_1 immediately (Rule G1). At time 2, ℓ_1 is requested by J_5, which is also enqueued in FQ_1 and suspended since it was non-empty. At time 4, $m = 2$ jobs hold the m-exclusion lock (*i.e.*, have entered FQ_1) and thus J_4 must enter PQ_1 instead (Rule G1). Hence it is safely deferred when ℓ_1 is later requested by higher-priority jobs (J_3, J_2, J_1). At the same time, J_5, which incurs pi-blocking until J_3's arrival at time 5, precedes the later-issued requests since it already held the m-exclusion lock—this avoids starvation in scenarios such as the one depicted in Figure 9(b). Note that J_5 incurs pi-blocking until time 5 (and not only until time 4) because the release of J_4 at time 4 does not displace J_5 from the set of the $c = m = 2$ highest-priority pending jobs (J_6 is also pending at time 4, but has a later deadline than J_5).

Next, we bound maximum s-oblivious pi-blocking under the OMLP's global mutex protcol. In the following analysis, let t_0 denote the time at which J_i issues $\mathcal{R}_{i,q,v}$, t_1 denote the time at which J_i enters FQ_q, and t_2 denote the time at which $\mathcal{R}_{i,q,v}$ is satisfied (this is illustrated in Figure 10 for J_4)

Further, let $entered(t)$, $t_0 \leq t < t_1$, denote the number of jobs that have been moved from PQ_q to FQ_q during $[t_0, t]$ due to Rule G3. That is, $entered(t)$ counts the jobs that preceded J_i in entering FQ_q. For example, for J_4 in Figure 10, $entered(5) = 0$, $entered(10) = 1$, and $entered(11) = 2$.

Lemma 13 *For each point in time $t \in [t_0, t_1)$, if J_i incurs s-oblivious pi-blocking, then $entered(t) < m$.*

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

Lemma 14 *During $[t_0, t_2)$, J_i incurs s-oblivious pi-blocking for the combined duration of at most $2 \cdot m - 1$ requests.*

Proof Due to the bounded length of FQ_q, at most $m - 1$ requests complete in $[t_1, t_2)$ before J_i's request is satisfied. By Lemma 13 and Rule G3, at most m requests complete before J_1 is no longer pi-blocked in $[t_0, t_1)$. □

Combining Lemma 14 with the maximum request length for each ℓ_q yields the following bound.

Lemma 15 *J_i is pi-blocked for at most*

$$b_i \triangleq \sum_{k=1}^{n_r} N_{i,q} \cdot (2 \cdot m - 1) \cdot L^{max} = O(m).$$

Proof By Lemma 14, J_i is pi-blocked for the duration of at most $2 \cdot m - 1$ requests each time it requests a resource ℓ_q. 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_q at most $N_{i,q}$ times, it suffices to consider the longest request $N_{i,k} \cdot (2 \cdot m - 1)$ times. The sum of the per-resource bounds yields b_i. By assumption (Section 3.2), $L^{max} = O(1)$ and $\sum_q N_{i,1} = O(1)$, and hence $b_i = O(m)$. □

A detailed, non-asymptotic bound on maximum pi-blocking that takes individual request lengths and frequencies into account is presented in Appendix A. Note that $b_i = 0$ if $N_{i,q} = 0$ for each ℓ_q, that is, the global OMLP is indeed independence-preserving.

4.6 Optimality, Combinations, and Limitations

In this section, we conclude our discussion of the OMLP family by examining various optimality properties and limitations in more detail and discuss how each of the protocol variants can be integrated with each other and OMLP-unrelated self-suspensions (such as suspensions due to I/O with dedicated devices).

4.6.1 Non-Asymptotic Optimality

Besides being asymptotically optimal, the four protocols of the OMLP family also have constant factors, summarized in Table 1, that are small enough for the protocols to be practical. Let $N_i = \sum_q N_{i,q}$ denote the maximum number of requests issued by any J_i. In the following, we assume $N_i > 0$, that is, the following discussion does not apply to independent tasks. From the example shown in Figure 2, it is apparent that a lower bound *per request* is $m - 1$ blocking requests. Therefore, a lower bound on the maximum number of blocking requests (under s-oblivious analysis) is $N_i \cdot (m - 1)$. This allows us to characterize how far the OMLP's bounds are from being optimal. Since L^{max} is presumed constant, we focus on the number of blocking requests in the following discussion.

Scheduling	Constraint	Progress Mechanism	Bound	Analysis
global	mutex	priority inheritance	$N_i \cdot (2m-1)$	Section 4.5
clustered	mutex	priority donation	$m + N_i \cdot (m-1)$	Section 4.2
clustered	k-exclusion	priority donation	$m + N_i \cdot \left\lceil \frac{m-\min\{k_q\}}{\min\{k_q\}} \right\rceil$	Section 4.4
clustered	RW—writers	priority donation	$2m + N_i \cdot (2m-1)$	Section 4.3
clustered	RW—readers	priority donation	$2m + N_i \cdot 2$	Section 4.3

Table 1 S-oblivious pi-blocking bounds of the OMLP given in terms of the maximum number of blocking requests

We begin with the global OMLP (for mutex constraints), which ensures that a job is pi-blocked by at most $N_i \cdot (2m-1)$ requests (see Table 1 for a summary of the OMLP's blocking bounds). As discussed in the preceding section, the principal advantage of the global OMLP over the clustered OMLP is that independent jobs do not incur pi-blocking (*i.e.*, if $N_i = 0$, then $b_i = 0$). The guaranteed upper bound is hence optimal within at most[7] a factor of

$$\frac{N_i \cdot (2m-1)}{N_i \cdot (m-1)} = \frac{N_i \cdot (2(m-1)+1)}{N_i \cdot (m-1)} = 2 + \frac{1}{m-1}.$$

That is, for large m, the global OMLP's bound on maximum s-oblivious pi-blocking is (almost) within a factor of two of the lower bound.

As summarized in Table 1, the clustered OMLP for mutex constraints ensures that a job J_i is pi-blocked by at most $m + N_i \cdot (m-1)$ conflicting requests. The mutex protocol is hence optimal within at most a factor of

$$\frac{m + N_i \cdot (m-1)}{N_i \cdot (m-1)} = 1 + \frac{m}{N_i \cdot (m-1)} \leq 1 + \frac{m}{(m-1)} = 2 + \frac{1}{m-1}.$$

The ratio is maximized for $N_i = 1$, in which case it approaches two for large m, just as the global OMLP. If $N_i > 1$, then the clustered OMLP ensures a smaller bound than the global OMLP, albeit at the cost of potentially delaying otherwise independent jobs.

In the case of the OMLP's k-exclusion protocol, if $k^{min} < m$, then Theorem 3 and Lemma 12 imply that the upper bound on s-oblivious pi-blocking is within at most a factor of

$$\begin{aligned}
\frac{\left(1 + \left\lceil \frac{m-k^{min}}{k^{min}} \right\rceil\right) + \left(N_i \cdot \left\lceil \frac{m-k^{min}}{k^{min}} \right\rceil\right)}{N_i \cdot \left\lceil \frac{m}{k^{min}} - 1 \right\rceil} &= \frac{\left(1 + \left\lceil \frac{m}{k^{min}} - 1 \right\rceil\right) + \left(N_i \cdot \left\lceil \frac{m}{k^{min}} - 1 \right\rceil\right)}{N_i \cdot \left\lceil \frac{m}{k^{min}} - 1 \right\rceil} \\
&= \frac{1}{N_i \cdot \left\lceil \frac{m}{k^{min}} - 1 \right\rceil} + \frac{1}{N_i} + 1 \\
&\leq 2 + \frac{1}{\left\lceil \frac{m}{k^{min}} - 1 \right\rceil}
\end{aligned}$$

of the lower bound that is unavoidable in the general case (for tasks that share resources). In the worst case, $k^{min} = 1$, the ratio reduces to $2 + \frac{1}{m-1}$. The k-exclusion protocol is thus no worse (in terms of the maximum number of blocking requests) than the clustered OMLP's mutex protocol.

In the degenerate case of $k^{min} = m$, maximum blocking under the clustered OMLP reduces to 1 (akin to a non-preemptive section), but the above ratio is undefined since the lower bound reduces to 0 in this case. This is because Lemma 12 does not take preemptions from higher-priority, later-arriving jobs into account. However, it is trivial to construct an example in which m lower-priority jobs request all k^{min} replicas such that a later-arriving, higher-priority job incurs s-oblivious pi-blocking for the duration of one critical section. The clustered OMLP is hence optimal in this case.

[7] It is unknown whether $N_i \cdot (m-1)$ is a tight lower bound in absolute terms (*i.e.*, non-asymptotically).

In the case of the OMLP's phase-fair RW lock, writers are delayed by additional requests because a reader phase may separate any two writer phases. This has the effect of essentially doubling the factor. That is, if job J_i issues N_i write requests (and no read requests), then the ensured bound is within at most

$$\begin{aligned}\frac{2m + N_i \cdot (2m-1)}{N_i \cdot (m-1)} &= \frac{2m + N_i \cdot (2(m-1)+1)}{N_i \cdot (m-1)} \\ &= 2 \cdot \left(1 + \frac{m}{N_i \cdot (m-1)}\right) + \frac{1}{m-1} \\ &\leq 2 \cdot \left(2 + \frac{1}{(m-1)}\right) + \frac{1}{m-1} \\ &= 4 + \frac{3}{m-1}\end{aligned}$$

of the optimal bound for mutex constraints. That is, for large m, the bound on maximum pi-blocking for (pure) writers approaches four and is hence approximately twice as large as the bounds of the mutex protocols. This suggests that RW locks should only be employed if the write ratio is small.

4.6.2 Optimality of Relaxed-Exclusion Protocols

Under phase-fair RW locks, read requests incur at most $O(1)$ acquisition delay. Similarly, requests for ℓ_q incur only $O(\frac{m}{k_q})$ acquisition delay under the k-exclusion protocol. Yet, we only prove $O(m)$ and $O(\frac{m}{k^{min}})$ *maximum* s-oblivious pi-blocking bounds, respectively—as discussed in Section 4.5, *any* job may become a priority donor and thus suspend (at most once) for the duration of the maximum request span. Since both relaxed-exclusion constraints generalize mutual exclusion, a priority donor might incur $\Omega(m)$ pi-blocking since *some* jobs might incur $\Omega(m)$ pi-blocking if $k^{min} = 1$ or if some resource is shared among m writers.

This seems undesirable for tasks that do not partake in mutual exclusion. For example, why should "pure readers" (*i.e.*, tasks that never issue write requests) not have an $O(1)$ bound on pi-blocking? It is currently unknown if this is even possible in general, as lower bounds for specific task types (*e.g.*, "pure readers," "DSP tasks") are an to-date unexplored topic that warrants further attention.

Since priority inheritance is sufficient for the global OMLP mutex protocol, one might wonder if it is possible to apply the same design using priority inheritance instead of priority donation to obtain an RW protocol under global scheduling with $O(m)$ maximum pi-blocking for writers and $O(1)$ maximum pi-blocking for readers. Unfortunately, this is not the case.

The reason is that the analytical benefits of priority inheritance under s-oblivious analysis do not extend to RW exclusion. When using priority inheritance with mutual exclusion, there is always a one-to-one relationship: a priority is inherited by at most one ready job at any time. In contrast, a single high-priority writer may have to "push" multiple low-priority readers. In this case, the high priority is "duplicated" and used by multiple jobs on different processors at the same time. This significantly complicates the analysis. In fact, simply instantiating Rules R1–R3 and W1–W3 from Section 4.3 with priority inheritance may cause $\Omega(\frac{n}{c})$ s-oblivious pi-blocking since it is possible to construct schedules that are conceptually similar to the one shown in Figure 3. A naive application of priority inheritance to the k-exclusion problem would lead to the same result.

This demonstrates the power of priority donation, and also highlights the value of the clustered OMLP even for the special cases $c = m$ and $c = 1$: the clustered OMLP RW and k-exclusion protocols are the first multiprocessor real-time locking protocols of their kind for the special cases of global and partitioned scheduling as well. In fact, to the best of our knowledge, no suspension-based RW protocol with $O(1)$ maximum pi-blocking for pure readers has been proposed to date.

In recent work [30], Elliot and Anderson presented a k-exclusion protocol for global JLFP schedulers that guarantees asymptotically optimal maximum s-oblivious pi-blocking while ensuring that independent jobs do not incur pi-blocking. Similar to the global OMLP, Elliot and Anderson's protocol uses priority inheritance in combination with a hybrid FIFO/priority queue. Due to the challenges of k-exclusion, their hybrid queue is of a more complicated structure than the one used in the global OMLP. Interestingly, Elliot and Anderson's protocol uses a technique akin to priority donation to ensure progress within each hybrid queue.

4.6.3 Highly Replicated Resources

Our k-exclusion protocol assumes $1 \leq k_q \leq m$ since additional replicas would remain unused as priority donation allows at most m incomplete requests. (The same assumption is made in Elliot and Anderson's k-exclusion protocol for global scheduling.) This has little impact on resources that require jobs to be scheduled (*e.g.*, shared data structures), but it may be a severe limitation for resources that do not require a processor (*e.g.*, there could be more than m DSP co-processors).

However, would a priority donation replacement that allows more than c jobs in a cluster to hold a replica be a solution? Surprisingly, the answer is no. This is because s-oblivious schedulability analysis (implicitly) assumes the number of processors as the maximum degree of parallelism (since all *pending* jobs are assumed to cause processor demand under s-oblivious analysis). In other words, s-aware schedulability analysis is required to derive analytical benefits from highly replicated resources. However, s-aware schedulability analysis poses additional challenges and, to the best of our knowledge, no k-exclusion protocol for s-aware analysis, optimal or otherwise, has been proposed to date.

4.6.4 Unrelated Self-Suspensions

An issue that arises in practice with real-time locking protocols is how blocking bounds are affected by suspensions that are unrelated to resource sharing. For example, a job may self-suspend when it performs I/O using a private device (*i.e.*, one that is not under control of a locking protocol). In uniprocessor locking protocols such as the SRP or the PCP, a job resuming from a self-suspension may incur additional pi-blocking just as if it were newly released. That is, a job J_i that self-suspends η_i times can incur pi-blocking for the duration of up to $1 + \eta_i$ outermost critical sections under the SRP or PCP.

This effect also applies to multiprocessor real-time locking protocols. For instance, under the MPCP and DPCP, a self-suspending job allows lower-priority jobs to issue requests for global resources and thus may incur additional pi-blocking after it resumes when lower-priority jobs are subsequently priority boosted.

Remarkably, the OMLP's blocking bounds are not affected by self-suspensions. In the case of the global OMLP, a job incurs pi-blocking only when it issues a request itself, which is not affected by locking-unrelated self-suspensions. Further, while it may appear on first sight that priority donation is affected by self-suspensions, this is not the case: a resuming job is never required to serve as a priority donor. This is because priority donation is defined in terms of *pending* jobs, and not in terms of *ready* jobs. A job that self-suspends or resumes does not alter the set of pending jobs. Further, any job serving as a priority donor upon release may self-suspend (since a priority donor's purpose is to suspend anyway). A priority donor that resumes from a self-suspension while the priority recipient executes is effectively not resumed until its donor services are no longer required.

The OMLP is hence not affected by self-suspensions and the presented analysis can be used in environments where jobs self-suspend. However, if jobs may self-suspend while holding a resource, then the maximum self-suspension time must be reflected in each $L_{i,q}$.

4.6.5 Protocol Combinations

The clustered mutex protocol (Section 4.2) generalizes the partitioned OMLP proposed in [17] in terms of blocking behavior; from an analytical point of view, there is thus little reason to use both in the same system or to prefer the partitioned OMLP over the more general clustered OMLP. However, in practice, it is somewhat easier to implement the partitioned OMLP since it relies on priority boosting instead of priority donation as a progress mechanism.

The clustered protocol variants can be freely combined since they all rely on priority donation and because their protocol rules do not conflict. However, care must be taken to correctly identify the maximum request span, which determines the maximum pi-blocking caused by priority donation.

The global OMLP cannot be used with any of the clustered OMLP variants since priority inheritance is incompatible with priority donation (from an analytical point of view). As discussed above, both the clustered and global mutex protocols have an $O(m)$ s-oblivious pi-blocking bound, but differ in constant factors and

with regard to which jobs incur pi-blocking. Specifically, since the global OMLP is independence-preserving, only jobs that request resources risk s-oblivious pi-blocking under the global OMLP, while even otherwise independent jobs may incur s-oblivious pi-blocking if they serve as a priority donor. The global OMLP may hence be preferable for $c = m$ if only few tasks share resources. We explore this tradeoff empirically in the following section.

5 Empirical Evaluation

The OMLP's defining feature is its asymptotic optimality under s-oblivious schedulablity analysis. However, asymptotic optimality does not necessarily translate into better schedulability in practice since it does not reflect constant factors. Further, one might reasonably suspect the s-oblivious analysis approach to be inherently too pessimistic to be of practical use. In other words, is the OMLP not just of theoretic interest, but also a good candidate for implementation in real-time operating systems?

To explore the OMLP's practical viability, we conducted large-scale schedulability experiments to empirically compare the OMLP with real-time locking protocols from prior work, and to contrast each of the OMLP's variants with each other. Specifically, we sought to answer the following questions.

- Is the OMLP competitive with previously-proposed s-oblivious locking protocols?
- Is the OMLP competitive with prior s-aware locking protocols?
- How do the two mutex protocols of the OMLP compare? Are both required?
- When does using the clustered OMLP's RW protocol offer an advantage (if any) over the clustered OMLP's mutex protocol, that is, is there an *analytical advantage* in allowing parallel reads compared to simply serializing both read and write requests?
- Does replicating shared resources significantly improve schedulability? That is, can resource replication be used to achieve a reduction in contention, and is this reduction reflected in the worst-case blocking analysis?
- Does k-exclusion offer an advantage over resource partitioning? In other words, when is the clustered OMLP's k-exclusion protocol preferable to statically assigning each task to one of the replicas?

In the following, we first describe the design of the experiments and then report upon our results and provide answers to the above questions in Sections 5.2–5.5.

5.1 Experimental Setup and Evaluation

A schedulablity experiment quantifies the performance of a real-time algorithm (such as a locking protocol) by determining the ratio of task sets that can be shown to be schedulable under it. The collection of task sets to be tested is typically generated randomly according to various parameter distributions. We followed this standard approach and generated task sets according to the following procedure, which is based on those previously used in [4, 11, 14, 18, 22].

5.1.1 Task Set Generation

Each task T_i was generated by drawing its period p_i uniformly from $\{10ms, 11ms, \ldots, 100ms\}$, by drawing its utilization u_i from a given utilization distribution, and by setting $e_i = u_i \cdot p_i$ (rounded to the next-largest microsecond). We considered three uniform, three exponential, and three bimodal utilization distributions.

- The ranges for the uniform distributions were [0.001, 0.1] (*light*), [0.1, 0.4] (*medium*), and [0.5, 0.9] (*heavy*).
- The mean of exponential distributions were 0.1 (*light*), 0.25 (*medium*), and 0.5 (*heavy*). The exponential distributions were further limited to the range (0, 1] by redrawing samples that did not fall within the range of feasible utilizations.
- In the three bimodal distributions, utilizations were distributed uniformly over either [0.001, 0.5) or [0.5, 0.9] with respective probabilities of 8/9 and 1/9 (*light*), 6/9 and 3/9 (*medium*), and 4/9 and 5/9 (*heavy*).

Given the number of processors m, a target *utilization cap* $U \in [1, m]$, and one of the nine utilization distributions, a random task set was generated by repeatedly creating tasks as described above until the task set's total utilization exceeded U, and by then reducing the last-added task's execution cost e_n such that U was reached exactly (*i.e.*, we ensured that $\sum_{i=1}^{n} u_i = U$). To avoid generating trivial task sets, we further ensured that each generated task set contained at least $m + 1$ tasks adding additional tasks if necessary (after reducing the execution cost of each task to ensure $\sum_{i=1}^{n} u_i = U$).

Each task's parameters $L_{i,q}^R$, $L_{i,q}^W$, $N_{i,q}^R$, and $N_{i,q}^W$ were randomly chosen based on the following parameters: the number of resources n_r, the *access probability* p^{acc}, the *write ratio* p^{write}, and a critical section length distribution. A given task T_i accessed resource ℓ_q with probability p^{acc} (*i.e.*, $P[N_{i,q} > 0] = p^{acc}$). If T_i accessed ℓ_q, then it was determined to be a writer with probability p^{write}, and a reader otherwise. The number of accesses $N_{i,q}$ was chosen uniformly from $\{1, \ldots, 5\}$. The maximum request length $L_{i,q}$ was chosen uniformly from one of three critical section length distributions. When using *short* critical sections, each $L_{i,q}$ was chosen randomly from $[1\mu s, 15\mu s]$; with *intermediate* critical sections, each $L_{i,q}$ was randomly chosen from $[1\mu s, 100\mu s]$; and finally, when assuming *long* critical sections, each critical section length was randomly chosen from $[5\mu s, 1280\mu s]$. To avoid generating implausible task sets, we ensured that the sum of all critical sections does not exceed a task's execution cost (*i.e.*, $\sum_{q=1}^{n_r} N_{i,q} \cdot L_{i,q} \leq e_i$) by reducing critical section lengths if necessary.

It is a widely acknowledged design principle that critical sections should be short: from a throughput point of view, long critical sections impede scalability, and from a real-time point of view, long critical sections result in pessimistic upper bounds on pi-blocking and thus limit schedulability. We therefore believe most critical sections to be short in practice. Nonetheless, in the schedulability study reported on in this paper, we also included intermediate critical section lengths to allow for pessimism when determining $L_{i,q}$ parameters in practice, and further considered long critical sections to allow for devices with inherently long critical sections (*e.g.*, GPUs). We chose the definition of long critical sections following Lakshmanan *et al.*, who assumed the stated range of critical section lengths in their evaluation of the MPCP and the MPCP-VS [36].

In the last step, if $c < m$, each generated task set was partitioned onto the $\frac{m}{c}$ clusters using the worst-fit decreasing heuristic. The worst-fit decreasing heuristic tends to spread the workload evenly across all clusters, thereby leaving a roughly equal amount of spare capacity in each cluster to compensate for utilization loss due to pi-blocking. In the case of global scheduling (*i.e.*, if $m = c$), partitioning is not required.

5.1.2 Tested Parameter Ranges

We conducted five separate schedulability studies using the described task set generation method to investigate the OMLP's performance in different scenarios. To avoid repetition, we summarize the parameter ranges common to all experiments in this section, and then describe each study's individual goal, setup, and results in the following sections in detail.

Unless noted otherwise, we considered each combination of the following parameter choices in each of the experiments. We considered three processor counts $m \in \{4, 8, 16\}$ and, for each processor count, three or four cluster sizes $c \in \{1, \frac{m}{4}, \frac{m}{2}, m\}$. The number of resources n_r was varied across $n_r \in \{\frac{m}{4}, \frac{m}{2}, m, 2m\}$. We let n_r depend on m based on the intuition that larger platforms are more likely to host complex workloads with a large number of shared resources.

In experiments focused on mutex or RW protocols, we let the access probability range across $p^{acc} \in \{0.1, 0.25, 0.4\}$. In experiments involving the clustered OMLP's k-exclusion protocol, we increased the access probability to $p^{acc} \in \{0.4, 0.55, 0.7\}$ following the intuition that resource replication is most appropriate if contention is inherently high.

In experiments involving the clustered OMLP's RW protocol, we considered $p^{write} \in \{0.05, 0.1, 0.2, 0.3, 0.5, 0.7\}$; all other studies pertaining to only mutex and k-exclusion protocols used $p^{write} = 1$.

As described in the discussion of the task set generation procedure, we considered short, medium, and long critical section lengths, and nine different utilization distributions. In total, this results in 3,564 parameter combinations (or *scenarios*) for mutex and k-exclusion experiments, and in 21,384 parameter combinations in experiments involving RW constraints (3,564 combinations for each choice of p^{write}). In each scenario, we

varied $U \in [\frac{m}{4}, m]$ in steps of 0.25 and generated and tested 1,000 task sets for each target utilization. In total, we generated more than 1,000,000,000 task sets over the course of the experiments reported in this paper.

5.1.3 Evaluation Criteria

Given the large number of considered scenarios, our schedulability data resulted in more than 40,000 graphs when visualized as standard schedulablity plots (*i.e.*, as a function of U, like those shown below in Figures 11–15). Our discussion of the results necessarily focuses on select example graphs since an exhaustive presentation of such a large data set is clearly infeasible. Nonetheless, we seek to report an *objective* aggregate view of the "big picture." We therefore chose to partially automate the evaluation by identifying each scenario in which a given protocol performs better than another protocol, where "performs better" is defined based on schedulability results with non-overlapping confidence intervals.

Let $S(\mathcal{A}, U)$ denote the fraction of task sets that could be claimed schedulable under the locking protocol (or algorithm) $\mathcal{A}$ for a given utilization cap U. For each tested $\mathcal{A}$ and each U, we determined the 95% bootstrap percentile confidence interval of $S(\mathcal{A}, U)$.[8] When comparing two locking protocols $\mathcal{A}_1$ and $\mathcal{A}_2$, we say that $\mathcal{A}_1$ provides *significantly higher schedulability* than $\mathcal{A}_2$ for a given U if and only if $S(\mathcal{A}_1, U) > S(\mathcal{A}_2, U)$ and the respective confidence intervals are disjoint.

Based on this definition of "better performing," and with respect to each pair of tested locking protocols $\mathcal{A}_1$ and $\mathcal{A}_2$, we classified each tested scenario into one of four groups.

1. Scenarios in which $\mathcal{A}_1$ *is clearly preferable*: $\mathcal{A}_1$ exhibits significantly higher schedulability for all or some of the tested utilization caps U and $\mathcal{A}_2$ does not perform significantly better than $\mathcal{A}_1$ for any U.
2. Analogously, scenarios in which $\mathcal{A}_2$ *is clearly preferable*.
3. Scenarios with *mixed results*: $\mathcal{A}_1$ and $\mathcal{A}_2$ both achieve significantly higher schedulability than the other for some (but not all) values of U.
4. Scenarios *without significant trends*: neither $\mathcal{A}_1$ nor $\mathcal{A}_2$ achieve significantly higher schedulability than the other for any value of U.

Having established precise evaluation criteria, we are now ready to present the key findings from our schedulability study. In addition to discussing select schedulability graphs (shown in Figures 11–15), we report the number of scenarios in each of the above categories in Tables 3–10 to provide some context for the highlighted examples. In particular, these categories allow us to objectively quantify whether the compared locking protocols perform well in "most" or only "few" of the tested scenarios. To the best of our knowledge, this is the first paper to report aggregate schedulability experiment results using an objective metric based on statistical significance.

5.2 Comparison of Mutex Protocols for Partitioned Scheduling

In the first schedulability experiment, we compared the OMLP to two previously-published real-time locking protocols to assess the practical viability of the OMLP and the s-oblivious approach.

For the case $1 < c < m$ and for RW or k-exclusion synchronization, there are no prior suspension-based real-time locking protocols to test against. However, the case $c = 1 < m$ (a partitioned multiprocessor system) has been the focus of much prior work on mutex protocols, and for this case, the combination of the MPCP [36, 42, 43] and P-FP scheduling is considered to be the de facto standard. For the MPCP, accurate

[8] Bootstrapping is a standard technique for estimating sampling statistics for unknown population distributions (*e.g.*, see [27]). Given a sample vector $X = (x_1, x_2, \ldots, x_s)$ consisting of s observations, N bootstrap sample vectors $Y^i = (y_1^i, y_2^i, \ldots, y_s^i)$, where $i \in \{1, \ldots, N\}$, are constructed by uniformly choosing each $y_k^i \in \{x_1, x_2, \ldots, x_s\}$ (*i.e.*, each Y^i is drawn from X with replacement). The distribution of a statistic $f(X)$ can then be estimated by applying f to each Y^i; an estimate of the 95%-confidence interval of $f(X)$ can be obtained from the 2.5$^{\text{th}}$ and 97.5$^{\text{th}}$ percentiles of the histogram of $f(Y^i)$. In our experiments, each x_k, where $1 \leq k \leq s = 1,000$, is a schedulability test result (*i.e.*, $x_k \in \{0, 1\}$) and the computed statistic is the sample mean (*i.e.*, the fraction of schedulable task sets). Bootstrapping is well-suited to schedulability experiments since it does not make any assumptions about the underlying population distribution. We used $N = 10,000$ bootstrap samples.

s-aware schedulability analysis exists [36]. In fact, the existing analysis of self-suspensions under uniprocessor FP scheduling (and hence also under P-FP scheduling) is arguably among the most accurate s-aware analysis published to date for any real-time scheduling policy. We therefore chose the MPCP to compare the OMLP's s-oblivious approach with the current state-of-the-art of s-aware schedulability analysis.

Besides presenting improved analysis of the original MPCP, Lakshmanan *et al*. also proposed a newer variant of the MPCP based on "virtual spinning" [36], where "spinning" jobs do in fact suspend, but other local jobs may not issue requests until the "spinning" job's request is complete. Notably, the analysis of this newer variant, which we denote as MPCP-VS, is s-oblivious. However, unlike the OMLP, the MPCP-VS does not ensure asymptotically optimal maximum pi-blocking since it uses priority queues to order conflicting requests. By comparing the OMLP with the MPCP-VS we can thus assess whether the OMLP's optimality leads to noticeable improvements in schedulability in practice.

Based on these considerations, we evaluated the following four combinations of locking protocol, scheduler, and type of schedulability analysis:

- the clustered OMLP's mutex protocol (Section 4.2) under P-EDF scheduling using the s-oblivious analysis presented in Appendix A.4;
- the clustered OMLP's mutex protocol under P-FP scheduling (using the same analysis);
- the MPCP [36, 42, 43] under P-FP scheduling using Lakshmanan *et al*.'s s-aware analysis [36]; and
- the MPCP-VS [36] under P-FP scheduling using Lakshmanan *et al*.'s s-oblivious analysis [36].

We generated task sets as described in Section 5.1.1 using the parameter combinations described in Section 5.1.2. However, we only considered clusters of size $c = 1$ since the published analysis of MPCP and MPCP-VS applies only to partitioned scheduling. Under P-EDF, we established the schedulability of each partition with Liu and Layland's classic s-oblivious utilization bound [37] after inflating each e_i by b_i (see Appendix A.7); under P-FP scheduling, we assigned rate-monotonic priorities [37] and applied uniprocessor response-time analysis [2, 35].

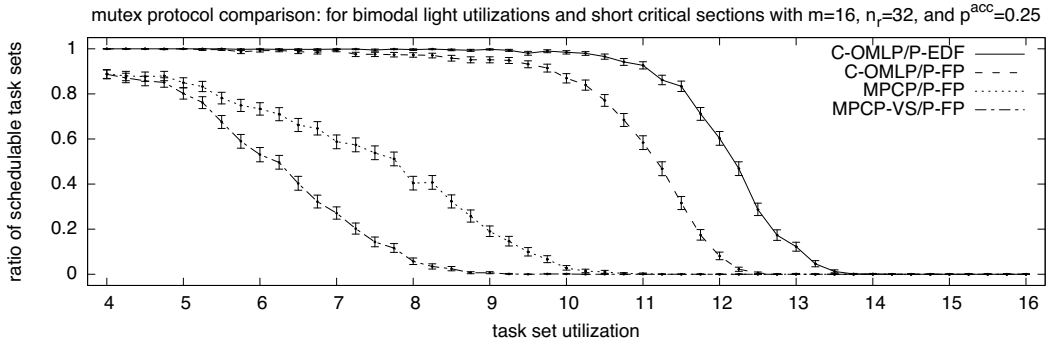
Note that there exists a cyclic dependency between the OMLP's blocking analysis and response-time analysis: to compute safe response times, a bound on worst-case pi-blocking is required, but a response-time bound is required to apply the blocking analysis presented in Appendix A. Under P-FP scheduling, we resolved this dependency using an iterative approach. Starting with the (clearly optimistic) assumption $r_i = e_i$, we alternatingly computed bounds on pi-blocking and then applied response-time analysis until reaching a fix-point for each task's response time (*i.e.*, all pi-blocking bounds were re-computed while the response-time of any task in any partition increased). Under P-EDF scheduling, we simply substituted each task's period p_i for its maximum response-time r_i (which is a safe, but likely pessimistic, bound since $r_i \leq p_i$ if the task set is deemed schedulable).

The described setup resulted in 972 schedulability graphs. Three representative examples are shown in Figure 11. We observed the following major trends.

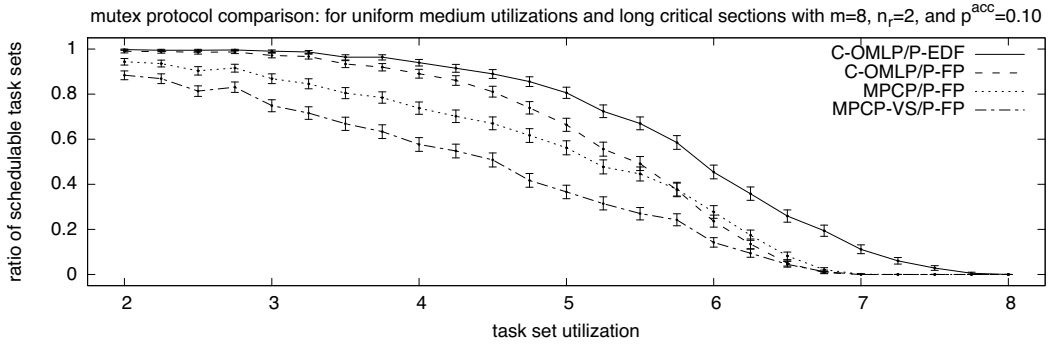
OMLP vs. MPCP-VS Figure 11(a) shows schedulability under each of the four configurations for bimodal light utilizations, short critical sections, $m = 16$, $n_r = 32$, and $p^{acc} = 0.25$. The error bars in this (and all subsequent figures) indicate 95% confidence intervals obtained using the bootstrap percentile method, as described in Section 5.1.3. It is immediately apparent that the OMLP under both P-FP and P-EDF provides significantly higher schedulability than either MPCP variant. We consider the s-oblivious MPCP-VS first.

The OMLP under P-FP scheduling clearly outperforms the MPCP-VS until $U \approx 12$. Since both configurations use the same schedulability test (*i.e.*, FP response-time analysis [2, 35]), rely on s-oblivious analysis, and have been applied to the same generated task sets, this provides strong evidence that the OMLP's asymptotic optimality translates into significantly improved pi-blocking bounds in practice. In fact, the OMLP outperformed the MPCP-VS in the vast majority of the tested scenarios, whereas the MPCP-VS did not provide significantly higher schedulability than the OMLP in *any* of the tested scenarios. This is apparent from Table 2, which reports scenario counts based on the classification defined in Section 5.1.3 and thus provides an objective summary of the entire data set.

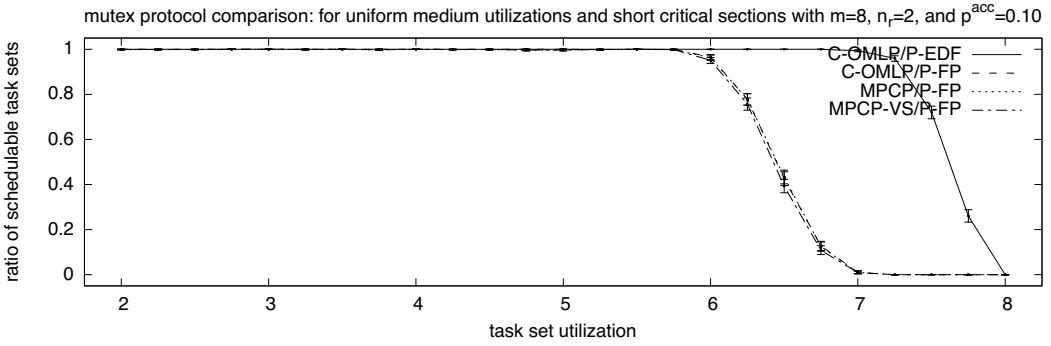
For example, the first row in Table 2 reveals that the OMLP was clearly preferable in 279 of the 324 scenarios with short critical sections length, one of which is shown in Figure 11(a). In contrast, the MPCP-VS



(a) Significantly higher schedulability under the OMLP



(b) Scenario in which neither the MPCP nor the OMLP under P-FP scheduling outerperform each other for all U



(c) Low-contention scenario in which the choice of locking protocol is irrelevant

Fig. 11 Comparison of mutex protocols: the clustered OMLP under P-EDF scheduling, the clustered OMLP under P-FP scheduling, the MPCP under P-FP scheduling and s-aware analysis, and the MPCP-VS under P-FP scheduling and s-oblivious analysis

was never preferable to the OMLP, and the experiment also did not yield any mixed results (*i.e.*, scenarios without clear trends). In 45 of the scenarios involving short critical sections, we did not observe any significant differences between the two protocols, which typically happens if there is only little contention for resources, in which case the choice of locking protocol becomes irrelevant. One such example is shown in Figure 11(c), which we revisit below. The counts of scenarios involving medium and long critical sections reveal that the OMLP outperforms the MPCP-VS in most of these cases as well. Overall, our results strongly suggest that the OMLP

critical sections	OMLP preferable	MPCP-VS preferable	mixed results	no trends
short	279	0	0	45
medium	307	0	0	17
long	250	0	0	74

Table 2 Summary of schedulability results: the clustered OMLP mutex protocol under P-FP scheduling with s-oblivious analysis vs. the MPCP-VS under P-FP scheduling with s-oblivious analysis (see Section 5.1.3 for definitions)

critical sections	OMLP preferable	MPCP preferable	mixed results	no trends
short	273	0	4	47
medium	300	0	4	20
long	238	0	12	74

Table 3 Summary of schedulability results: the clustered OMLP mutex protocol under P-FP scheduling with s-oblivious analysis vs. the MPCP under P-FP scheduling with s-aware analysis (see Section 5.1.3 for definitions)

provides superior schedulability in a wide range of scenarios, and also validate the intuition underlying maximum pi-blocking, namely that it is a useful complexity metric that reflects meaningful performance characteristics.

OMLP vs. MPCP While the MPCP performs somewhat better than the MPCP-VS in the scenario depicted in Figure 11(a), it still provides significantly lower schedulability than the OMLP. In this particular example, the MPCP reaches zero schedulability already at $U \approx 10$, whereas the OMLP does so only at $U \approx 12$ under P-FP scheduling, and only at $U \approx 13.5$ under P-EDF scheduling. While the exact margin in schedulability differs from scenario to scenario, this example is representative of a larger trend, as evidenced by the scenario counts provided in Table 3. Similarly to the MPCP-VS, the MPCP was not preferable to the OMLP in any of the tested scenarios. In contrast, the OMLP provided equal or superior schedulability in the vast majority of the tested scenarios: across all critical section lengths, the OMLP was clearly preferable in 811 out of the 972 tested scenarios. Notably, a few scenarios revealed mixed results, in the sense that both the MPCP and the OMLP under P-FP scheduling were significantly better for some, but not all, U. One such example is shown in Figure 11(b), which depicts schedulability for uniform medium utilizations, long critical sections, $m = 8$, $n_r = 2$, and $p^{acc} = 0.10$. While there are only few requests per job in this scenario (there are only few resources with low access probability), schedulability under each of the tested locking protocols is affected by the long critical sections. The OMLP under P-FP scheduling provides significantly higher schedulability than the MPCP in the range $U \in [2, 5.5]$. However, with increasing U, schedulability decreases under the OMLP somewhat faster than under the MPCP, with the result that the MPCP provides (slightly) higher schedulability in the range $[6.25, 6.75]$. Since neither the OMLP nor the MPCP achieve equal or higher schedulability for *all* values of U, this scenario is counted as having mixed results.

Nonetheless, the aggregate results reported in Table 3 document that there exist many scenarios in which the OMLP provides better schedulability than the de-facto standard protocol for partitioned P-FP scheduling. This shows that the OMLP in particular, and the s-oblivious analysis approach in general, have practical merit and can in fact provide superior schedulability results. This also suggests that future evaluations of real-time locking protocols should consider both s-oblivious and s-aware alternatives—s-oblivious protocols are not necessarily more pessimistic than s-aware locking protocols.

One might wonder whether this is primarily due to the OMLP's design, or whether this mostly reflects limitations in existing s-aware analysis. We believe the OMLP's competitiveness to be a result of both; however, given that the employed analysis of suspensions under uniprocessor FP scheduling [2, 36] is already rather accurate, we do not expect this observation to be invalidated in the foreseeable future.

MPCP vs. MPCP-VS Although our main goal was to compare the OMLP to both MPCP variants, our data set also allows for a comparison of the s-aware MPCP with the s-oblivous MPCP-VS. In fact, our results paint a

critical sections	MPCP preferable	MPCP-VS preferable	mixed results	no trends
short	210	0	0	114
medium	246	0	0	78
long	192	0	0	132

Table 4 Summary of schedulability results: the MPCP under P-FP scheduling with s-aware analysis vs. the MPCP-VS under P-FP scheduling with s-oblivious analysis (see Section 5.1.3 for definitions)

critical sections	P-EDF preferable	P-FP preferable	mixed results	no trends
short	323	0	0	1
medium	317	0	0	7
long	265	0	0	59

Table 5 Summary of schedulability results: P-EDF scheduling with the clustered OMLP vs. P-FP scheduling with the clustered OMLP (see Section 5.1.3 for definitions)

very clean picture: the MPCP-VS performs significantly worse than the MPCP in about two thirds of the tested scenarios, with the remainder of the scenarios not showing a significant trend (see Table 4). This is illustrated by the scenarios shown in insets (a) and (b) of Figure 11, where the MPCP-VS performs significantly worse than the other tested protocols in the entire range where meaningful differences exist (*i.e.*, before schedulability reaches zero under all protocols).

The observation that the MPCP-VS performs significantly worse than the MPCP in most cases confirms earlier results reported by Lakshmanan *et al.* [36], who came to a similar conclusion using a different experimental setup and a different performance metric. In future work, there is thus little reason to consider the MPCP-VS instead of better-performing alternatives such as the OMLP in case of s-oblivious analysis, or the classic MPCP [42, 43] and the FMLP$^+$ [14] in case of s-aware analysis.

P-FP vs. P-EDF One of the advantages of the OMLP family is that it is compatible with any JLFP scheduler. The analysis presented in Appendix A therefore applies to both P-EDF and P-FP scheduling. As one might expect, we found that the OMLP under P-EDF scheduling almost always achieves significantly higher schedulability than the OMLP under P-FP scheduling, despite using *identical* analysis. The difference in schedulability is entirely due to the optimality of EDF on uniprocessors.

An example of this effect is shown in Figure 11(c), which shows schedulability for the same scenario as shown in inset (b) with short instead of long critical sections. Since contention is low, pi-blocking is equally low under each of the tested locking protocols. As a result, there are no significant differences among the locking protocols under P-FP scheduling (as indicated by overlapping confidence intervals), whereas the OMLP under P-EDF scheduling provides significantly higher schedulability despite nearly identical bounds on pi-blocking. The OMLP under P-EDF is similarly the best-performing configuration in the scenarios depicted in insets (a) and (b) of the same figure. Overall, the OMLP under P-EDF is clearly the best-performing configuration in 905 of the 972 tested scenarios (see Table 5), and frequently by a large margin. The few scenarios in which no significant differences could be observed are due to parameter combinations that result in excessive contention (especially if long critical sections are involved), which causes equally low schedulability under all considered locking protocols and schedulers.

While it is hardly surprising that schedulability is generally higher under P-EDF than under P-FP, there is an important point to be made: the OMLP under P-FP scheduling is mainly interesting in the context of "apples-to-apples" comparisons with other locking protocols for P-FP scheduling. If the goal is instead to maximize schedulability, then the OMLP under P-EDF is clearly the better choice.

In summary, our schedulability study comparing mutex protocols found that, in a large majority of the scenarios, the OMLP under P-FP scheduling performed better than either the MPCP or the MPCP-VS. Further, the OMLP under P-EDF scheduling performed better than any of the locking protocols under P-FP scheduling;

there is thus little reason to favor P-FP scheduling over P-EDF scheduling from a locking point of view. The MPCP-VS never outperformed any of the other locking protocols in any of the tested scenarios. It is worth emphasizing that the point of these experiments is *not* to claim that the OMLP and s-oblivious analysis are *always* superior. Rather, our results show that there exist (many) scenarios in which the OMLP and s-oblivious analysis are a practical alternative that is often competitive with, and at times even superior to, established s-aware locking protocols.

This concludes our comparison of the OMLP with previously-proposed alternatives. In the experiments discussed in the remainder of this section, we compared the OMLP family's four protocols with each other to determine when each variant is most appropriate.

5.3 Comparison of the OMLP's Mutex Protocols for Global and Clustered Scheduling

In the second schedulability study, we compared the two mutex protocols of the OMLP family for global and clustered scheduling. Since global scheduling is a special case of clustered scheduling, it is not obvious that both protocols are needed. In fact, there are two ways in which each protocol could conceivably be sufficient by itself:

- if the clustered OMLP always yields pi-blocking bounds comparable to those of the global OMLP, then there is little reason to support the global OMLP as well; and, conversely,
- if it is possible to partition task sets such that each resource is shared only *within* each cluster, and not across cluster boundaries, then a global locking protocol (applied to each cluster) may suffice.

To test whether either possibility is in fact the case, we conducted a schedulability experiment using the task set generation method described in Section 5.1.1. We compared the clustered OMLP, which permits *inter-cluster locking* (*i.e.*, the sharing of resources across cluster boundaries), against the global OMLP instantiated within each cluster, which only permits *intra-cluster locking* (*i.e.*, resources may only be shared among tasks assigned to the same cluster). Both protocols were applied on top of C-EDF scheduling; we did not consider P-FP scheduling in this experiment.

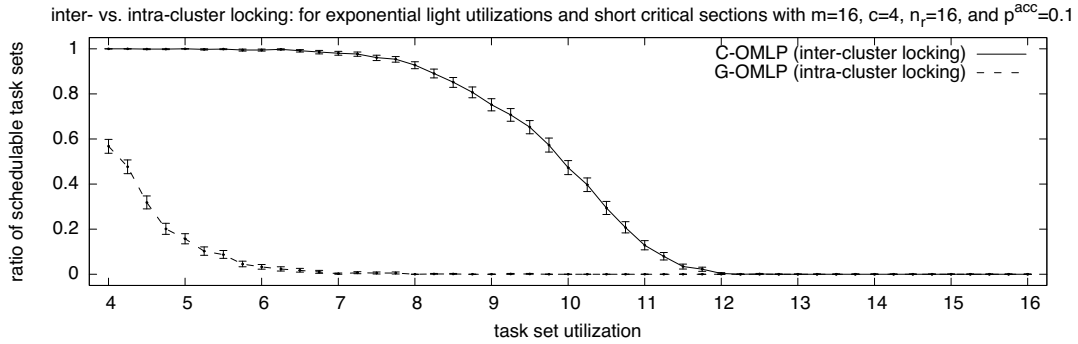
Under the clustered OMLP, if $c < m$, each generated task set was partitioned on a task-by-task basis using the worst-fit decreasing heuristic as described in Section 5.1.1. After computing bounds on pi-blocking using the analysis presented in Section A.4 assuming $r_i = p_i$, we inflated each e_i with b_i in accordance with the s-oblivious analysis approach (Section A.7). A task set was deemed schedulable if each cluster passed at least one of three standard s-oblivious G-EDF schedulability tests [7, 12, 33] (if $c > 1$) or if none of the clusters was over-utilized (if $c = 1$).

Under the global OMLP, an additional partitioning constraint must be enforced if $c < m$ since only intra-cluster locking is permitted. That is, instead of assigning individual tasks to clusters, groups of tasks sharing resources must be assigned as a whole. Consider the *resource-sharing graph* G obtained by creating a vertex for each task, and by creating an edge between any two tasks sharing some resource. To satisfy the intra-cluster locking constraint, the partitioning phase must assign each connected component of G to one of the clusters (without over-utilizing any of the clusters).

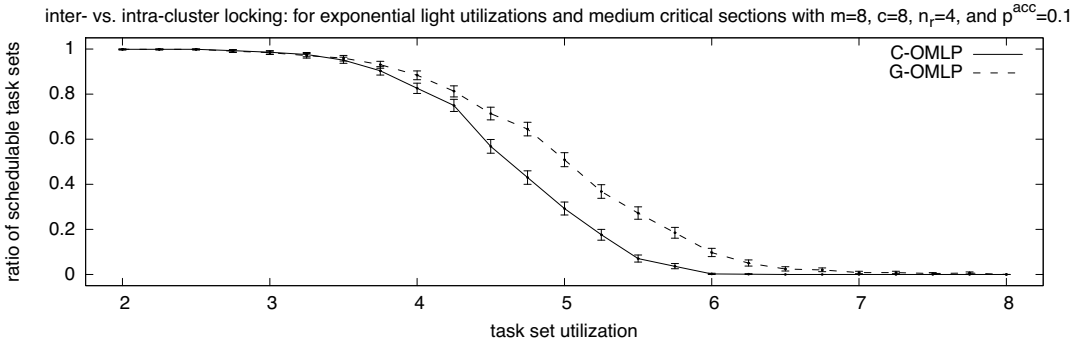
If a valid assignment of connected components to clusters could be found (or if $c = m$), bounds on pi-blocking were obtained by applying the analysis presented in Section A.3 to the subset of tasks in each cluster (assuming $r_i = p_i$). The schedulability of each cluster was established using the same s-oblivious schedulability tests as in the case of the clustered OMLP. If no valid assignment of connected components to clusters could be found using the worst-fit decreasing heuristic, then the task set was deemed unschedulable.

Intra- vs. inter-cluster locking We evaluated both approaches using all 3,564 parameter combinations from Section 5.1.2, including each combination of c and m. Our results clearly show that neither locking protocol is always better than the other; rather, for each variant, there exist scenarios in which it provides a significant advantage over the alternative.

The graph shown in Figure 12(a) corresponds to a scenario in which the clustered OMLP is clearly preferable to the global OMLP (within each cluster). In the depicted example with exponential light utilizations, short



(a) Applying a global locking protocol within each cluster fails if clusters are small



(b) Under global scheduling ($m = c$), the global mutex protocl is not limited by partitioning constraints

Fig. 12 Comparison of the OMLP's mutex protocols for global and clustered scheduling under C-EDF scheduling

critical sections, $m = 16$, $c = 4$, $n_r = 16$, and $p^{acc} = 0.10$, schedulability under the global OMLP is close to zero even for small utilization caps, whereas the clustered OMLP achieves schedulability close to one until $U \approx 8$ and reaches zero schedulability only at $U \approx 12$. The reason for this disparity is the difficulty of partitioning large connected components. In the depicted scenario, each cluster of size $c = 4$ is relatively small compared to the total number of processors $m = 16$. Further, since the light utilization distribution results in low average per-task utilizations, the task set size n grows rapidly with increasing utilization caps U. Consequently, the average connected component size also grows as U increases, and it hence becomes increasingly difficult, or even impossible, to partition task sets such that resources are not shared across cluster boundaries. Schedulability under the G-EDF is correspondingly low.

However, there also exist scenarios in which the global OMLP performs better than the clustered OMLP. One such case is shown in Figure 12(b), which depicts schedulability under global scheduling ($m = c$) with exponential light distributions, medium critical sections, $n_r = 4$, and $p^{acc} = 0.1$. Since partitioning is not required under global scheduling, the performance of the global OMLP is not impacted by the intra-cluster locking constraint. While the gap in observed schedulability is not as large as in inset (a), the global OMLP does provide significantly higher schedulability for a wide range of utilization caps.

The two discussed examples illustrate that neither the clustered nor the global OMLP are superfluous. In fact, the aggregate summary of the entire data set, reported in Table 6, reveals that it is not even possible to clearly state which protocol is preferable "most" of the time. In addition to the fact that we observed mixed results (*i.e.*, significant differences, but no clear winner) in more than 900 scenarios, each protocol is favored by different critical section lengths. In the case of short critical sections, the clustered OMLP is clearly preferable in 690 scenarios, whereas the global OMLP is only preferable in 40 scenarios. In contrast, the clustered OMLP is

critical sections	inter-cluster locking	intra-cluster locking	mixed results	no trends
short	690	40	341	117
medium	598	109	376	105
long	308	381	253	246

Table 6 Summary of schedulability results: the OMLP's mutex protocol for clustered scheduling (applied across clusters using priority donation) vs. the OMLP's mutex protocol for global scheduling (applied within each cluster using priority inheritance)

less often preferable than the global OMLP in the case of long critical sections (308 scenarios vs. 381 scenarios, respectively). What is the cause for this shift in relative performance?

Recall from Section 4 that the two locking protocols use different progress mechanisms and hence have different pi-blocking characteristics (summarized in Table 1). On the one hand, jobs incur pi-blocking for the duration of at most $m - 1$ critical sections per request under the clustered OMLP, whereas jobs may incur pi-blocking for up to $2m - 1$ critical sections per request under the global OMLP. On the other hand, the clustered OMLP uses priority donation, which has the disadvantage that a job may incur pi-blocking upon release while it serves as a priority donor. In contrast, the global OMLP is based on priority inheritance and is hence independence-preserving—jobs incur pi-blocking only due to contention for resources that they request.

Due to the lower per-request bound, tasks that issue more than one request are likely to have lower pi-blocking bounds under the clustered OMLP. However, the clustered OMLP may not be a feasible choice if there exists at least one task that cannot tolerate pi-blocking due to priority donation (for example, if the task has only little slack or a very short period). In fact, the results summarized in Table 6 show that the clustered OMLP becomes less competitive (relative to the global OMLP) as critical section lengths increase, which suggests that priority donation may indeed be the limiting factor in these scenarios. To test this hypothesis, we conducted a third schedulability experiment with a modified task set generation procedure.

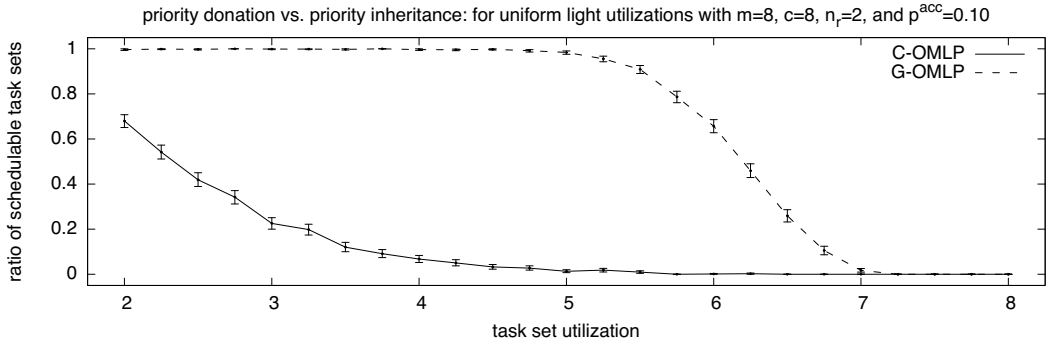
Heterogeneous task sets To simulate task sets with a wide variety of temporal constraints, we modified the task set generation procedure to produce less uniform task sets. Instead of generating all tasks based on the same distributions, we created two classes of tasks, namely *urgent tasks* and *obstructing tasks*.

Obstructing tasks were created in large parts as described in Section 5.1.1; however, the critical section lengths of obstructing tasks was drawn exclusively from the long critical section distribution. In contrast, urgent tasks were configured to have only short critical sections. Further, the period of urgent tasks was chosen uniformly from $\{3ms, 4ms, \ldots, 33ms\}$. Finally, we ensured that each resource was shared among either only obstructing or only urgent tasks, but not across task classes. By design, urgent tasks are thus sensitive to pi-blocking (due to their short periods), but are not *directly* exposed to long critical sections.

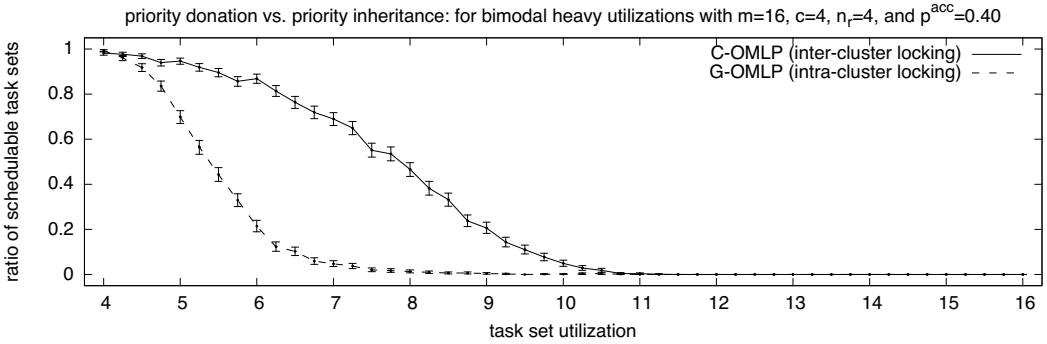
To control the generation of urgent and obstructing tasks, we introduced an additional *urgent fraction* parameter, denoted f_u. Based on f_u, the utilization cap U was divided such that the total utilization of urgent tasks equaled $U \cdot f_u$, with the remaining capacity consumed by obstructing tasks. We considered $f_u \in \{\frac{1}{4}, \frac{1}{2}, \frac{3}{4}\}$ in our experiment.

Priority donation vs. priority inheritance We repeated the comparison of the clustered OMLP and the global OMLP (instantiated within each cluster) as described above using the heterogeneous task set generation procedure. As expected, the heterogeneous task sets proved to be much more difficult to schedule, and especially so under the clustered OMLP.

Figure 13(a) depicts schedulability under each mutex protocol in the case of global scheduling with uniform light utilizations, $m = 8$, $n_r = 2$, and $p^{acc} = 0.1$. Because priority donation exposes urgent tasks to delays due to the long critical sections of obstructing tasks, schedulability is severely limited under the clustered OMLP. In contrast, the global OMLP achieves high schedulability because no partitioning is required (since $c = m$) and because jobs are not pi-blocked by unrelated requests under it. This demonstrates the value of using an independence-preserving locking protocol if some tasks are sensitive to delays and others have (excessively)



(a) Schedulability is low under the clustered OMLP because urgent tasks are not isolated from long critical sections



(b) Schedulability is low under the global OMLP because the intra-cluster locking constraint cannot be satisfied for large n

Fig. 13 Comparison of the OMLP's mutex protocols for global and clustered scheduling under C-EDF scheduling in the case of heterogeneous task sets

long critical sections. In other words, priority inheritance is the only viable progress mechanism in this case as both priority donation and priority boosting create pi-blocking dependencies among all tasks.

However, instantiating the global OMLP within each cluster still requires partitioning on the granularity of connected components. In the case of small clusters, satisfying the intra-cluster locking constraint is often more limiting than the negative effects of priority donation. This is apparent in Figure 13(b), which shows much reduced schedulability under the global OMLP in the case of bimodal heavy utilizations with $m = 16$, $c = 4$, $n_r = 4$, and $p^{acc} = 0.4$. Due to the high access probability and the low number of resources, the resource sharing graph G is likely to degenerate into a single connected component—in this case, applying the global OMLP is infeasible for task sets that require more than $c = 4$ processors. Thus, while schedulability under the clustered OMLP may not be particularly good in this difficult scenario, it is still significantly higher than under the global OMLP. The clustered OMLP is hence clearly preferable in this example.

When considering the entire data set as a whole (see Table 7), it becomes apparent that the *relative cluster size* is the performance-determining factor in the case of heterogeneous task sets. If $1 \leq c < \frac{m}{2}$, each cluster is small compared to the total number of processors; partitioning connected components is therefore difficult in this case and schedulability under the global OMLP is severely limited. Otherwise, if $\frac{m}{2} \leq c \leq m$, partitioning is either not required (under global scheduling) or comparably easy (there are only two large clusters if $c = \frac{m}{2}$), which provides a major advantage to the global OMLP. This is obvious from the scenario counts reported in Table 7: in scenarios with relatively small clusters, the clustered OMLP (based on priority donation) is more than twice as often preferable than global OMLP, but in scenarios with two large clusters or under global scheduling,

relative cluster size	priority donation	priority inheritance	mixed results	no trends
$1 \leq c < \frac{m}{2}$	807	309	404	100
$\frac{m}{2} \leq c \leq m$	2	1685	179	78

Table 7 Summary of schedulability results: the OMLP's mutex protocol for clustered scheduling (applied across clusters using priority donation) vs. the OMLP's mutex protocol for global scheduling (applied within each cluster using priority inheritance)

the global OMLP (and hence priority inheritance) is clearly preferable in the vast majority of the tested scenarios (in more than 1,600 scenarios out of roughly 1950 scenarios in this category).

Overall, our experiments show that neither the global nor the clustered OMLP are ideal if some degree of independence among (subsets of) tasks must be maintained under clustered scheduling with $c < m$. However, this is not a limitation specific to the OMLP—rather, it shows that there is a need for an asymptotically optimal, independence-preserving locking protocol with support for inter-cluster resource sharing. We plan to study this issue in greater detail in future work.

5.4 Comparison of the Clustered OMLP's Mutex and RW Protocols

In the fourth schedulablity experiment, we explored under which conditions the clustered OMLP's RW protocol is preferable to its mutex counterpart. We used the (homogeneous) task set generation method described in Section 5.1.1 to compare both protocols under C-EDF scheduling. Analogous to the experiments discussed in the previous section, we accounted for pi-blocking under the mutex and RW protocols using the analysis presented in Section A.4 and Section A.5, respectively, and then checked the temporal correctness of each cluster using s-oblivious schedulability tests. We evaluated both protocols in all 21,384 scenarios resulting from the parameter combinations specified in Section 5.1.2, including $p^{write} \in \{0.05, 0.1, 0.2, 0.3, 0.5, 0.7\}$.

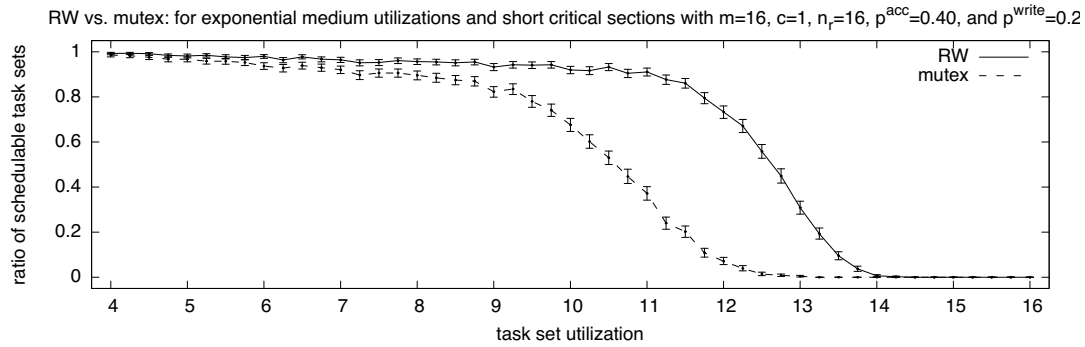
Obviously, an RW lock should yield improved schedulability if most requests are read requests. However, if this is not the case—if writes are frequent—then schedulability can in fact be worse under phase-fair RW locks than under regular mutex locks. While the clustered OMLP's RW protocol ensures *asymptotically* optimal maximum pi-blocking for writers, its bounds are subject to constant factors that are about twice as large as those of the mutex protocol bounds (recall Section 4.6 and Table 1). That is, in the worst case, a writer may incur twice as much pi-blocking under phase-fair RW locks since it can be delayed by both $m - 1$ previously-issued writes and m interleaved reader phases. In contrast, under the clustered OMLP's mutex protocol, a writer is delayed by at most $m - 1$ previously-issued requests of any kind.

Fundamentally, phase-fairness is a tradeoff between greatly reduced pi-blocking for readers and (possibly) doubled pi-blocking for writers, which is only beneficial if writes are much less frequent than reads. It is therefore expected that the write ratio p^{write} strongly impacts the relative performance of the two locking protocols. Our results confirm that this is indeed the case.

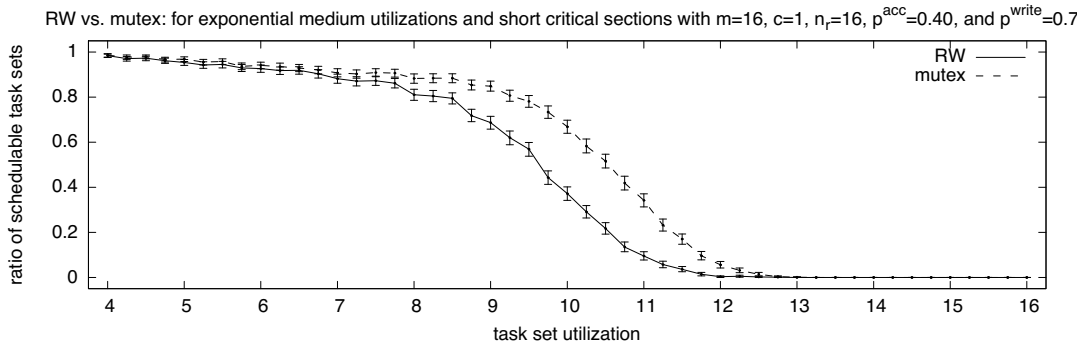
An example scenario in which the write ratio is suitably low ($p^{write} = 0.2$) is shown in Figure 14(a), which shows schedulability under each of the two protocols for exponential medium utilizations, short critical sections, $m = 16$, partitioned scheduling, $n_r = 16$, and $p^{acc} = 0.4$. In this case, delaying writers in favor of (frequent) reads is a valid tradeoff. As a result, schedulability is significantly higher under the RW protocol than under its mutex counterpart over a wide range of utilization caps. The RW protocol is clearly preferable in this case.

In contrast, inset (b) depicts an example in which the assumption underlying the RW protocol is violated since, on average, more than two thirds of the requests are writes. Aside p^{write}, all parameters in the scenario depicted in inset (b) are the same as in the scenario depicted in inset (a). While the mutex protocol is not affected by p^{write} (all requests are serialized anyway), schedulability under the RW protocol is markedly lower, to the point that the mutex protocol is clearly preferable in this case—the relative performance of the mutex protocol has improved (without changing in absolute terms) because schedulability under the RW protocol has decreased.

This effect is in fact representative of the entire data set. Table 8 reports the scenario counts for each tested write ratio. If $p^{write} \leq 0.10$, the mutex protocol is never preferable since it unnecessarily serializes reads.



(a) The clustered OMLP's RW protocol is preferable if writes are infrequent



(b) The clustered OMLP's mutex protocol is preferable if writes are common

Fig. 14 Comparison of the clustered OMLP's mutex and RW protocols under C-EDF scheduling

p^{write}	RW preferable	mutex preferable	mixed results	no trends
0.05	2876	0	0	688
0.10	2758	0	0	806
0.20	2493	19	14	1038
0.30	2128	91	23	1322
0.50	539	314	1	2710
0.70	0	1343	0	2221

Table 8 Summary of schedulability results: the clustered OMLP's RW protocol vs. the clustered OMLP's mutex protocol

However, as the write ratio increases, the mutex protocol becomes more competitive. While the mutex protocol is preferable only in a few extreme scenarios if $p^{write} = 0.20$ or $p^{write} = 0.30$, it already provides higher schedulability in more than 300 scenarios if $p^{write} = 0.50$, although the RW protocol remains preferable in more than 500 of the tested scenarios. Finally, in the most extreme of the tested write ratios ($p^{write} = 0.70$), we did not find the RW protocol to be preferable in any of the tested scenarios, while the mutex protocol provided significantly higher schedulability in over 1,300 scenarios. However, it is not the case that the RW protocol is necessarily worse than the mutex protocol if the write ratio is high. In many cases where there is little to be gained from RW locks, the RW protocol simply reduces to mutex-like performance, as evidenced by the large number of scenarios in the "no trends" column.

Overall, the schedulability experiment confirmed the intuition underlying RW locks: the use of the clustered OMLP's RW protocol likely results in improved schedulability if the write ratio is low, and has no effect, or even a negative effect, if this requirement is not met. This matches our earlier experience with spin-based phase-fair RW locks [18].

5.5 Comparison of the Clustered OMLP's Mutex and k-Exclusion Protocols

In the fifth and final schedulability experiment, we compared the clustered OMLP's mutex and k-exclusion protocols under C-EDF scheduling. We sought to explore two questions: does adding replicas improve schedulability significantly? And how should replicas be managed?

With regard to the latter question, there are two fundamental ways that a set of tasks may share k_q replicas of a resource, as previously discussed in Section 4.4. Recall that either each task may access any of the replicas, in which case a k-exclusion protocol is required, or that each task is statically assigned one of the replicas, which are then managed individually using a mutex protocol. The former approach corresponds to global processor scheduling, whereas the latter approach, which we refer to as the *p-mutex* approach, is akin to partitioned scheduling. When using the p-mutex approach, a heuristic is required to assign tasks to resources. Since the goal is to minimize contention, requests should be spread out evenly among the replicas. We therefore simply assigned tasks to replicas in a round-robin manner, which ensured that each replica was used by roughly the same number of tasks. In future work, it may be interesting to explore more-sophisticated replica assignment heuristics.

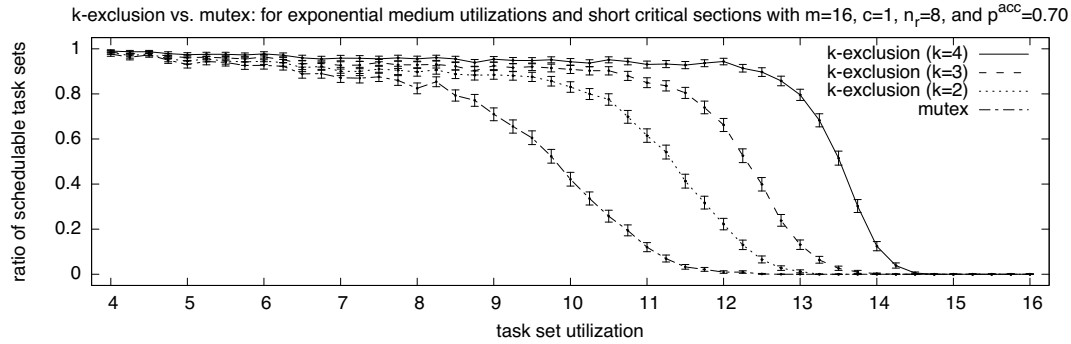
With regard to partitioning and schedulability testing, we used an experimental setup analogous to the one described in the previous section. We compared seven different combinations of locking protocol and degree of replication k_q:

- the clustered OMLP's mutex protocol (without resource replication) as a baseline;
- three configurations of the clustered OMLP's k-exclusion protocol, one for each $k_q \in \{2, 3, 4\}$; and
- three configurations of the p-mutex approach using the clustered OMLP's mutex protocol to serialize accesses to each replica, similarly with $k_q \in \{2, 3, 4\}$.

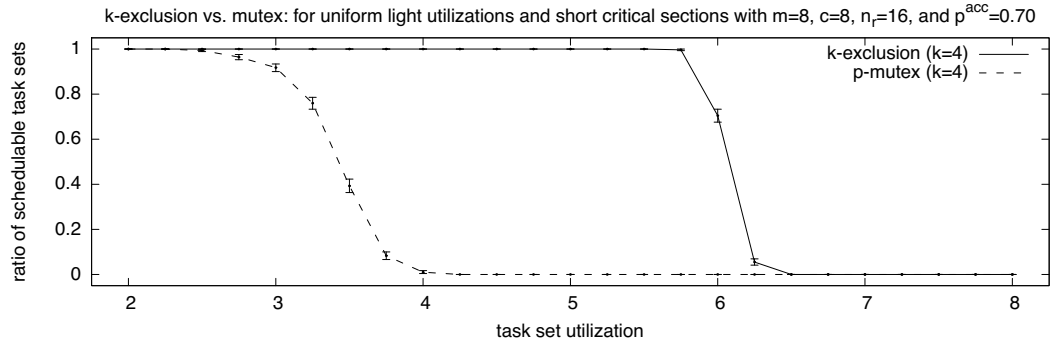
Note that all resources were assumed to have a uniform replication factor (*e.g.*, in the configuration of the k-exclusion protocol assuming $k_q = 3$, *each* resource was assumed to be replicated three times). While this is unlikely to be the case in practice, this approach allows us to isolate the effects of resource replication.

We evaluated each configuration under each of the parameter combinations from Section 5.1.2. Since plotting seven curves per graph would result in too much clutter, we plotted four graphs corresponding to each scenario: one graph showing the mutex protocol in comparison with each of the three configurations of the k-exclusion protocol, and one graph for each tested k_q comparing the k-exclusion protocol to its corresponding p-mutex configuration.

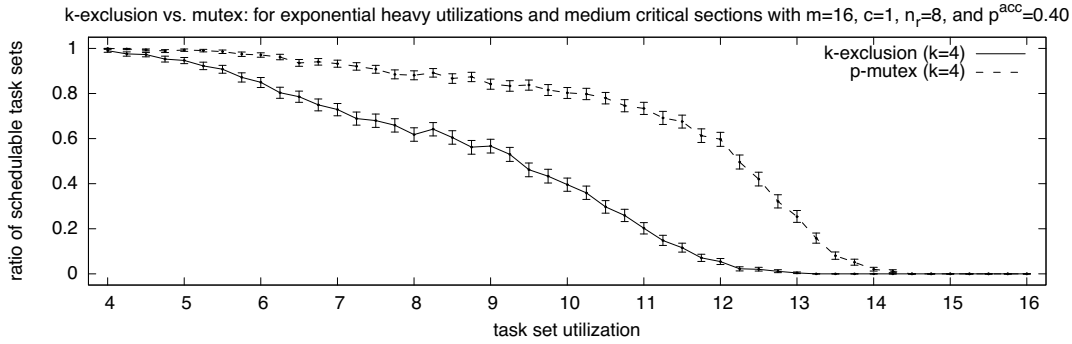
Replication benefits A graph of the former kind for the case of exponential medium utilizations, short critical sections, $m = 16$, $c = 1$, $n_r = 8$, and $p^{acc} = 0.7$ is shown in Figure 15(a). In the depicted scenario, contention is high due to the high access probability. As expected, schedulability increases significantly with each added replica. This example shows that k-replication of resources can be a very effective measure to improve schedulability, an observation that is representative of most of the considered scenarios. Table 9 provides a summary of the effect of replicating each resource k_q times compared to using the clustered OMLP's mutex protocol without resource replication. Reassuringly, the data shows that the OMLP's k-exclusion protocol is able to exploit resource replication to achieve lower bounds pi-blocking in most scenarios, and that the OMLP's k-exclusion protocol is never worse than using the mutex protocol (without replication) instead. As might be expected, the data further shows that the number of scenarios in which the k-exclusion protocol is clearly preferable increases with the number of replicas. We conclude that adding replicas is not only effective at reducing average contention at runtime, but also a potent measure for reducing worst-case contention, and thus effective at improving schedulability.



(a) Adding replicas improves schedulability



(b) The OMLP's k-exclusion protocol is preferable if contention is high and the ratio m/k_q is small



(c) The p-mutex approach is preferable if contention is low and m is large

Fig. 15 Comparison of the clustered OMLP's mutex and k-exclusion protocols under C-EDF scheduling

P-mutex vs. k-exclusion Insets (b) and (c) of Figure 15 show graphs comparing the clustered OMLP's k-exclusion protocol with the p-mutex approach for $k_q = 4$. Surprisingly, the two graphs show conflicting trends. Figure 15(b) depicts a high-contention scenario with uniform light utilizations and short critical sections under global scheduling for $m = 8$, $n_r = 16$, and $p^{acc} = 0.70$. Since task sets are relatively large (due to the very low per-task utilizations), and since each task accesses many resources (due to the high access probability and the large number of shared resources), each resource is shared by more than $m \cdot k_q$ tasks for large U. Consequently, under the p-mutex approach, each replica is assigned at least m tasks, which implies that the analytical advantage

k_q	k-replication preferable	mutex preferable	mixed results	no trends
2	2053	0	0	1511
3	2498	0	0	1066
4	2866	0	0	698

Table 9 Summary of schedulability results: the clustered OMLP's k-exclusion protocol vs. the clustered OMLP's mutex protocol (without replication)

utilizations	m	k-exclusion preferable	p-mutex preferable	mixed results	no trends
light	4	280	0	0	44
	8	222	121	1	88
	16	127	161	7	137
medium	4	213	0	0	111
	8	86	219	7	120
	16	9	366	3	54
heavy	4	106	0	0	218
	8	5	277	2	148
	16	0	372	0	60

Table 10 Summary of schedulability results: the clustered OMLP's k-exclusion protocol (for $k_q = 4$) vs. the clustered OMLP's mutex protocol (each task is statically assigned to one of the $k_q = 4$ replicas)

of replication is lost since the clustered OMLP's mutex per-request bound of $m - 1$ blocking requests becomes the limiting factor. In contrast, the analysis of the OMLP's k-exclusion protocol is capable of providing an analytical advantage despite the high contention because its analysis considers all replicas as a whole and can thus reflect the increased completion rate (recall Lemma 11). The OMLP's k-exclusion protocol is hence clearly preferable to the p-mutex approach in this scenario.

However, as is apparent in Figure 15(c), this is not always the case. The depicted scenario shows that the p-mutex approach provides significantly higher schedulability in the case of heavy exponential utilizations, medium critical sections, $m = 16$, $c = 1$, $n_r = 8$, and $p^{acc} = 0.40$. In this scenario, contention is much lower than in the scenario depicted in inset (b) since the heavy utilization distribution results in smaller task set sizes, and also because the access probability is lower. In this case, statically assigning tasks to replicas results in, on average, fewer than m tasks sharing each replica. Consequently, the bounds on pi-blocking under the p-mutex approach are lower than under the k-exclusion protocol. This suggests that it may be worthwhile to study in future work whether the bounds on pi-blocking under the OMLP's k-exclusion protocol can be tightened, or whether a different queue structure would yield a k-exclusion protocol that is always preferable to the p-mutex approach.

With the existing analysis, a mixed picture emerges when considering the entire data set. Table 10 reports the scenario counts for the comparison of the p-mutex approach with the clustered OMLP's k-exclusion protocol for the case of $k_q = 4$. The table is structured based on the number of processors m and the "weight" of the employed per-task utilization distributions (*i.e.*, the uniform light, exponential light, and bimodal light utilization distributions are reported together as "light utilizations", *etc.*). The data reveals two major trends. First, the k-exclusion protocol performs best for small m, or rather, if the ratio m/k_q is small. This is apparent both from the fact that the p-mutex approach is never clearly preferable to the k-exclusion protocol if $m = 4$, and also from the fact that, with respect to each utilization distribution weight, the number of scenarios in which the k-exclusion protocol is clearly preferable decreases with increasing m. The second major trend is that the k-exclusion protocol performs best for light utilization distributions (where there are many tasks and contention is high), whereas the p-mutex approach is much more competitive for heavy utilization distributions. This provides strong quantitative support for the above explanation of the trends seen in insets (b) and (c) of Figure 15.

Overall, our results show that in many scenarios there is clear value in supporting a dedicated k-exclusion protocol, but that one should not assume that the k-exclusion protocol is always less pessimistic than a static task-to-replica assignment. Perhaps not coincidentally, this matches the situation in real-time multiprocessor scheduling, where neither partitioned nor global scheduling is always preferable, either.

5.6 Limitations and future directions

We have presented a large-scale, thorough empirical evaluation of the OMLP family of locking protocols. Our results show that the OMLP compares well with both the s-aware MPCP and the s-oblivious MPCP-VS, and also that none of the protocols in the OMLP family is redundant, in the sense that each protocol excels at certain parameter combinations and task set compositions. However, as with any experimental study, and although we made an effort to consider a large range of diverse scenarios and configurations, there are some interesting aspects that had to remain beyond the scope of our study (which is already quite large and time-consuming, and in fact required several weeks of processor time on a large compute cluster). We briefly mention two issues that would be interesting to consider in more detail in future work.

For one, no attempt was made to take resource-sharing considerations into account when partitioning task sets. That is, tasks were assigned using a worst-fit decreasing heuristic based on each task's utilization. Better results could likely be achieved by employing resource-sharing-aware partitioning heuristics. However, such heuristics are still an area of active research [34, 36, 41] and beyond the scope of this paper. It would be interesting to reevaluate the relative performance of the considered locking protocols once it has become clear which partitioning heuristics are best suited to partitioning task sets with shared resources.

We also did not account for implementation overheads in the comparison of the locking protocols. In an actual implementation, semaphore-based locking protocols incur potentially high overheads because kernel support is typically required to implement priority inheritance, priority boosting, and priority donation. Further, suspensions are generally costly because jobs lose cache affinity while suspended. As a result, spinlocks are often more efficient for short critical sections [14].

We recognize the importance of considering implementation overheads and acknowledge that it would be interesting to incorporate real-world overheads into our schedulability experiments. Nonetheless, we chose to omit such overheads from the study presented in this paper for the following reasons. First, each of the compared locking protocols in this paper is a suspension-based locking protocol, which means that each protocol is likely impacted equally by implementation overheads. That is, while the *absolute* performance of each tested locking protocol would be affected by overheads, the performance of the locking protocols *relative* to each other would likely not change even if overheads were included. And, second, to account for real-world overheads, actual overheads must be collected and analyzed in a real system, which constrains the combinations of c and m that can be tested to those available in present lab machines. Since we are primarily interested in algorithmic differences in this paper, we instead chose to vary c and m freely to explore a larger range of possible platforms. However, we note that we have implemented the MPCP, the MPCP-VS, and a prototype of priority donation in LITMUS$^{\mathrm{RT}}$. Experiments similar to those presented in this paper for the case $m = 24$ under consideration of overheads can be found in [14].

6 Conclusion

We have provided a general, precise definition of pi-blocking in suspension-based locking protocols and proposed maximum pi-blocking as a natural measure of a locking protocol's blocking behavior. We identified two classes of commonly-used schedulability analysis, namely s-oblivious and s-aware analysis, and showed a lower bound on maximum s-oblivious pi-blocking of $\Omega(m)$.

We have shown this bound to be asymptotically tight by designing the first suspension-based multiprocessor real-time locking protocols with $O(m)$ maximum pi-blocking under s-oblivious pi-blocking. We showed that in the special case of mutex constraints under global scheduling, optimal pi-blocking can be achieved with priority

inheritance. To achieve optimality in the other cases (*i.e.*, RW and k-exclusion, or if $1 < c < m$), we designed a new form of restricted priority boosting, named priority donation, which is a novel mechanism for ensuring resource-holder progress that works for $1 \leq c \leq m$.

We have designed, analyzed, and empirically evaluated the OMLP family of protocols for mutual, RW, and k-exclusion. The OMLP family is asymptotically optimal under s-oblivious analysis under any JLFP scheduler with arbitrary cluster sizes. The two relaxed-exclusion protocols have the desirable property that the reduction in contention is reflected analytically in improved worst-case acquisition delays ($O(1)$ for readers and $O(\frac{m}{k_q})$ in the k-exclusion case, compared to $O(m)$ for all jobs under mutex locks). The clustered OMLP's mutex protocol is the first of its kind for clustered scheduling with $1 < c < m$; the RW and k-exclusion protocols are further the first of their kind for the special cases of partitioned and global scheduling as well.

We conducted a large-scale schedulability study involving more than one billion task sets to assess the OMLP's practical viability. To objectively report the results from the more than 40,000 generated graphs, we developed a classification system based on statistical significance that allows us to summarize trends from a large number of evaluated parameter configurations without requiring manual intervention (and without introducing human bias). Using this new methodology, we made three key observations: the s-oblivious analysis approach is practical and not necessarily more pessimistic than existing s-aware analysis; the OMLP is competitive with, and often superior to, the s-aware MPCP and the s-oblivious MPCP-VS; and there is no single best choice for all scenarios. Rather, each of the protocols in the OMLP family has specific advantages that make it the best-performing protocol in certain scenarios. Taken together, the OMLP family provides high schedulability in a wide range of scenarios.

Besides the empirical work remarked upon in Section 5.6, a number of intriguing algorithmic challenges remain to be addressed in future work, including support for fine-grained locking and nested critical sections, optimal locking protocols for clustered scheduling under s-aware analysis, tight lower bounds on s-aware and s-oblivious maximum pi-blocking under RW and k-exclusion protocols, and the design of optimal RW and k-exclusion protocols for both types of analysis.

Acknowledgements We thank Tomas Kalibera for suggesting the use of bootstrapping in schedulability experiments. Work supported by the Max Planck Society; NSF grants CNS 0834270, CNS 0834132, and CNS 1016954; ARO grant W911NF-09-1-0535; AFOSR grant FA9550-09-1-0549; and AFRL grant FA8750-11-1-0033. The first author was supported in part by a UNC Dissertation Completion Fellowship.

References

1. Andersson, B., Easwaran, A.: Provably good multiprocessor scheduling with resource sharing. Real-Time Systems **46**(2), 153–159 (2010)
2. Audsley, N., Burns, A., Richardson, M., Tindell, K., Wellings, A.: Applying new scheduling theory to static priority pre-emptive scheduling. Software Engineering Journal **8**(5), 284–292 (1993)
3. Baker, T.: Stack-based scheduling for realtime processes. Real-Time Systems **3**(1), 67–99 (1991)
4. Baker, T.: A comparison of global and partitioned EDF schedulability tests for multiprocessors. Tech. Rep. TR-051101, Florida State University (2005)
5. Baker, T., Baruah, S.: Schedulability analysis of multiprocessor sporadic task systems. In: Handbook of Real-Time and Embedded Systems. Chapman Hall/CRC (2007)
6. Baker, T., Baruah, S.: Sustainable multiprocessor scheduling of sporadic task systems. In: Proceedings of the 21st Euromicro Conference on Real-Time Systems, pp. 141–150 (2009)
7. Baruah, S.: Techniques for multiprocessor global schedulability analysis. In: Proceedings of the 28th IEEE Real-Time Systems Symposium, pp. 119–128 (2007)
8. Baruah, S., Burns, A.: Sustainable scheduling analysis. In: Proceedings of the 27th IEEE Real-Time Systems Symposium, pp. 159–168 (2006)
9. Bastoni, A.: Towards the integration of theory and practice in multiprocessor real-time scheduling. Ph.D. thesis, Universita' degli Studi di Roma "Tor Vergata" (2011)

10. Bastoni, A., Brandenburg, B., Anderson, J.: An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers. In: Proceedings of the 31st IEEE Real-Time Systems Symposium, pp. 14–24 (2010)
11. Bastoni, A., Brandenburg, B., Anderson, J.: Is semi-partitioned scheduling practical? In: Proceedings of the 23rd Euromicro Conference on Real-Time Systems, pp. 125–135 (2011)
12. Bertogna, M., Cirinei, M.: Response-time analysis for globally scheduled symmetric multiprocessor platforms. In: Proceedings of the 28th IEEE Real-Time Systems Symposium, pp. 149–160 (2007)
13. Block, A., Leontyev, H., Brandenburg, B., Anderson, J.: A flexible real-time locking protocol for multiprocessors. In: Proceedings of the 13th IEEE Conference on Embedded and Real-Time Computing Systems and Applications, pp. 47–57 (2007)
14. Brandenburg, B.: Scheduling and locking in multiprocessor real-time operating systems. Ph.D. thesis, The University of North Carolina at Chapel Hill (2011)
15. Brandenburg, B., Anderson, J.: A comparison of the M-PCP, D-PCP, and FMLP on LITMUS$^{\mathrm{RT}}$. In: Proceedings of the 12th International Conference On Principles Of Distributed Systems, LNCS 5401, pp. 105–124. Springer-Verlag (2008)
16. Brandenburg, B., Anderson, J.: An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS$^{\mathrm{RT}}$. In: Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 185–194 (2008)
17. Brandenburg, B., Anderson, J.: Optimality results for multiprocessor real-time locking. In: Proceedings of the 31st Real-Time Systems Symposium, pp. 49–60 (2010)
18. Brandenburg, B., Anderson, J.: Spin-based reader-writer synchronization for multiprocessor real-time systems. Real-Time Systems **46**(1), 25–87 (2010)
19. Brandenburg, B., Anderson, J.: Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In: Proceedings of the 9th ACM International Conference on Embedded Software (2011)
20. Brandenburg, B., Calandrino, J., Block, A., Leontyev, H., Anderson, J.: Synchronization on real-time multiprocessors: To block or not to block, to suspend or spin? In: Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 342–353 (2008)
21. Calandrino, J., Anderson, J., Baumberger, D.: A hybrid real-time scheduling approach for large-scale multicore platforms. In: Proceedings of the 19th Euromicro Conference on Real-Time Systems, pp. 247–256 (2007)
22. Calandrino, J., Leontyev, H., Block, A., Devi, U., Anderson, J.: LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In: Proceedings of the 27th IEEE Real-Time Systems Symposium, pp. 111–123 (2006)
23. Carpenter, J., Funk, S., Holman, P., Srinivasan, A., Anderson, J., Baruah, S.: A categorization of real-time multiprocessor scheduling problems and algorithms. In: Handbook of Scheduling: Algorithms, Models, and Performance Analysis. Chapman Hall/CRC (2004)
24. Chen, C., Tripathi, S.: Multiprocessor priority ceiling based protocols. Tech. Rep. CS-TR-3252, Univ. of Maryland (1994)
25. Chen, M., Lin, K.: A priority ceiling protocol for multiple-instance resources. In: Proceedings of the 12th IEEE Real-Time System Symposium, pp. 140–149 (1991)
26. Courtois, P., Heymans, F., Parnas, D.: Concurrent control with "readers" and "writers". Communications of the ACM **14**(10), 667–668 (1971)
27. Davison, A., Hinkley, D.: Bootstrap methods and their application. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press (1997)
28. Devi, U., Leontyev, H., Anderson, J.: Efficient synchronization under global EDF scheduling on multiprocessors. In: Proceedings of the 18th Euromicro Conference on Real-Time Systems, pp. 75–84 (2006)
29. Easwaran, A., Andersson, B.: Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In: Proceedings of the 30th IEEE Real-Time Systems Symposium, pp. 377–386 (2009)
30. Elliott, G., Anderson, J.: An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems. In: Proceedings of the 19th International Conference on Real-Time and Network Systems (2011)

31. Faggioli, D., Lipari, G., Cucinotta, T.: The multiprocessor bandwidth inheritance protocol. In: Proceedings of the 22nd Euromicro Conference on Real-Time Systems, pp. 90–99 (2010)
32. Gai, P., di Natale, M., Lipari, G., Ferrari, A., Gabellini, C., Marceca, P.: A comparison of MPCP and MSRP when sharing resources in the Janus multiple processor on a chip platform. In: Proceedings of the 9th IEEE Real-Time And Embedded Technology Application Symposium, pp. 189–198 (2003)
33. Goossens, J., Funk, S., Baruah, S.: Priority-driven scheduling of periodic task systems on multiprocessors. Real-Time Systems **25**(2-3), 187–205 (2003)
34. Hsiu, P.C., Lee, D.N., Kuo, T.W.: Task synchronization and allocation for many-core real-time systems. In: Proceedings of the 9th ACM international conference on Embedded software, pp. 79–88. ACM (2011)
35. Joseph, M., Pandya, P.: Finding response times in a real-time system. The Computer Journal **29**(5), 390–395 (1986)
36. Lakshmanan, K., Niz, D., Rajkumar, R.: Coordinated task scheduling, allocation and synchronization on multiprocessors. In: Proceedings of the 30th IEEE Real-Time Systems Symposium, pp. 469–478 (2009)
37. Liu, C., Layland, J.: Scheduling algorithms for multiprogramming in a hard real-time environment. Journal of the ACM **30**, 46–61 (1973)
38. Liu, J.: Real-Time Systems. Prentice Hall (2000)
39. Macariu, G., Cretu, V.: Limited blocking resource sharing for global multiprocessor scheduling. In: Proceedings of the 23rd Euromicro Conference on Real-Time Systems, pp. 262–271 (2011)
40. Nemati, F., Behnam, M., Nolte, T.: Independently-developed real-time systems on multi-cores with shared resources. In: Proceedings of the 23rd Euromicro Conference on Real-Time Systems, pp. 251–261 (2011)
41. Nemati, F., Nolte, T., Behnam, M.: Partitioning real-time systems on multiprocessors with shared resources. In: Proceedings of the 14th International Conference on Principles of Distributed Systems, LNCS 6490, pp. 253–269 (2010)
42. Rajkumar, R.: Real-time synchronization protocols for shared memory multiprocessors. Proceedings of the 10th International Conference on Distributed Computing Systems pp. 116–123 (1990)
43. Rajkumar, R.: Synchronization In Real-Time Systems—A Priority Inheritance Approach. Kluwer Academic Publishers (1991)
44. Rajkumar, R., Sha, L., Lehoczky, J.: Real-time synchronization protocols for multiprocessors. Proceedings of the 9th IEEE Real-Time Systems Symposium pp. 259–269 (1988)
45. Ridouard, F., Richard, P., Cottet, F.: Negative results for scheduling independent hard real-time tasks with self-suspensions. In: Proceedings of the 25th IEEE Real-Time Systems Symposium, pp. 47–56 (2004)
46. Schliecker, S., Negrean, M., Ernst, R.: Response time analysis on multicore ECUs with shared resources. IEEE Transactions on Industrial Informatics **5**(4), 402–413 (2009)
47. Sha, L., Rajkumar, R., Lehoczky, J.: Priority inheritance protocols: an approach to real-time synchronization. IEEE Transactions on Computers **39**(9), 1175–1185 (1990)

A Schedulability Analysis

In this appendix, we first introduce a generic framework for expressing bounds on pi-blocking and then apply it to bound pi-blocking under each of the locking protocols presented in this paper. The blocking analysis presented in the following is essential for deriving safe blocking bounds suitable for schedulability analysis. However, such bounds tend to be somewhat technical in nature and are primarily required only for implementing schedulability tests (as used in the schedulability experiments presented in Section 5); the casual reader may safely skip this appendix and consult the overview presented in Section 4 instead.

The framework presented in the following is generic in the sense that it is not tied to any particular locking protocol. It serves two purposes. For one, it avoids redundancy in the subsequent analysis of the locking protocols, which have structurally similar blocking bounds. Second, the presented analysis takes a *holistic* analysis approach to reduce the pessimism inherent in analyzing requests individually. That is, it is intended to be applied to each job as a whole and bounds blocking across *all* requests that a job issues, instead of bounding delays on a request-by-request basis.

The presented holistic analysis approach was first used to analyze the FMLP under P-FP scheduling [16], and subsequently further developed to analyze RW spinlocks [18]. The version presented herein has been somewhat simplified compared to the previous variants. We next explain the intuition underlying the approach, which we then formalize in Section A.2 below.

A.1 Holistic Blocking Analysis

In the following, let J_i denote an arbitrary job of the task T_i for which a bound on maximum blocking is being derived. The main idea of the holistic approach is to avoid accounting for any individual possibly-blocking request more than once, and to avoid accounting for requests that cannot possibly interfere with J_i's requests. In particular, when a job request the same resource more than once, the holistic approach can avoid substantial pessimism compared to analyzing each resource request in isolation, and especially so if long requests occur much less frequently than short requests (*i.e.*, if there are large differences among the tasks' $L_{i,q}$, $N_{i,q}$, and p_i parameters).

Example 9 To illustrate possible pessimism when analyzing requests individually, consider the following scenario (in this and the following examples, the use of the clustered OMLP's mutex variant is assumed). Suppose a task T_i shares a serially-reusable resource ℓ_q with another task T_x. Further, suppose J_i requests ℓ_q up to $N_{i,q} = 20$ times and that jobs of T_x hold ℓ_q for at most $L_{x,q} = 10$ time units. Finally, suppose jobs of T_x require ℓ_q at most once while any J_i is pending. When analyzing each of J_i's many requests individually (*i.e.*, when bounding the maximum pi-blocking incurred by a single request), T_x's sole interfering request is effectively considered to block *each* of J_i's requests since T_x's request might delay *any* of the requests (but not all at once). Consequently, J_i's overall bound on pi-blocking due to requests for ℓ_q would be $N_{i,q} \cdot L_{x,q} = 200$ time units, whereas the actual maximum possible delay is only $L_{x,q} = 10$ time units since, when considering J_i's entire execution, J_x obviously delays J_i with at most only one blocking request for ℓ_q, and not with up to 20.

This example demonstrates that maximum contention should be analyzed as a whole across all of J_i's requests for a particular resource. (Since we assume that requests for resources are not nested, blocking bounds for individual resources are independent from each other and can be derived individually.) The extent to which J_i is blocked due to requests for a resource ℓ_q in the worst case is limited by the following constraints:

1. *Maximum number of requests issued by other jobs.* As discussed above in Example 9, if jobs of T_x issue at most k requests while any J_i is pending, then J_i will be blocked by at most k requests of jobs of T_x, regardless of the number of requests issued by J_i.
2. *Maximum number of interfering requests per request issued by J_i.* Suppose J_i requests a serially-reusable resource ℓ_q only once, that $m = 4$, and that ℓ_q is requested by other jobs up to $k = 100$ times while J_i is pending. In this case, J_i is delayed by at most $m - 1 = 3$ competing requests, irrespective of the total number of requests k for ℓ_q since priority donation limits the maximum queue length to m jobs.
3. *Maximum number of interfering requests per task.* For example, suppose ℓ_q is shared among three tasks T_i, T_x, and T_y. If J_i issues only one request, then it is blocked by at most one request from T_x and one request from T_y, irrespective of the total number of requests issued by these tasks, and irrespective of the number of processors. Due to the FIFO ordering in the wait queue FQ_q, each task can precede J_i at most once per request.
4. *Task locality.* For example, suppose T_i shares a resource with tasks T_x and T_y under partitioned scheduling ($c = 1$), and that T_i and T_x are assigned to processor 1, whereas T_y is assigned to processor 2. Jobs of T_y can cause J_i to incur acquisition delay because they can issue conflicting requests while J_i is scheduled. In contrast, jobs of T_x cannot cause J_i to incur acquisition delay because jobs of T_x are not scheduled while J_i is executing; however, a job J_x can cause J_i to incur pi-blocking if J_i must serve as J_x's priority donor upon release.

We formalize these four constraints next.

A.2 Interference Sets

We begin with Constraint 1 by bounding the maximum resource requirements of competing tasks. In the task model assumed in this paper, a task T_i's resource requirements are characterized by the parameters $N_{i,q}$ and $L_{i,q}$. The main advantages of this model are that it is general enough to reflect many possible job behaviors (*e.g.*, no particular request order or minimum separation of requests is assumed) and that the required information can be obtained as part of worst-case execution time analysis (or empirically bounded if such analysis is not available). However, it is possible that more detailed knowledge is available for specific applications.

For example, it could be the case that jobs of a task T_i access a resource ℓ_q twice, and that the second access is always much shorter than the first access. In this case, using a single upper bound $L_{i,q}$ for both requests is needlessly pessimistic. A similar concern arises with resources that are not accessed by *every* job of T_i. For, example to reduce overheads, an application T_a could be programmed to record status information in a shared log ℓ_l only once every five jobs. Assuming that each J_a requests access to ℓ_l would needlessly overestimate contention for ℓ_l. However, explicitly incorporating all such considerations yields a task model that is overly complicated for our goal (which is to study the underlying algorithmic properties of the protocols).

We instead use an abstraction called *task interference bound* to achieve a separation of concerns between the modeling of resource requirements and the actual analysis of locking protocols, which is structurally independent from model considerations. A task's interference bound (for non-processor resources) is similar to a demand bound function (for processor time) in that it "upper bounds" a task's worst-case resource requirement during some interval. The actual blocking analysis is expressed in terms of task interference, which can be defined to take advantage of detailed application-specific resource usage information. The primary benefit of this approach is that derived blocking terms can be reused to derive less pessimistic bounds when additional information in form of a more-detailed task model is available.

In the following, to achieve the desired separation of concerns, we formalize a task's "interference bound" as a set of requests that safely approximates a task's "actual contention" for a resource. Recall from Section 2.2 that $\mathcal{R}_{i,q,v}$ denotes the v^{th} request for resource ℓ_q issued by any J_i, and that $\mathcal{L}_{i,q,v}$ denotes the request length of $\mathcal{R}_{i,q,v}$. This allows us to formalize the concept of a task's "contention for a resource."

Definition 6 Suppose jobs of a task T_i execute k resource requests for a resource ℓ_q during an interval $[t_0, t_1)$. In a concrete, fixed schedule, the *contention* due to T_i during $[t_0, t_1)$ is the set of requests

$$C_{i,q}(t_0, t_1) \triangleq \left\{\mathcal{R}_{i,q,v}, \mathcal{R}_{i,q,(v+1)}, \ldots, \mathcal{R}_{x,q,(v+k-1)}\right\}$$

such that $\mathcal{R}_{i,q,v}$ is the first request and $\mathcal{R}_{i,q,(v+k-1)}$ is last request issued by any J_i during $[t_0, t_1)$.

In general, v and k are unknown prior to the execution of T_i, as is the length of each request in $C_{i,q}(t_0, t_1)$. To enable *a priori* analysis, a generic notion of worst-case contention is required. The purpose of T_i's request interference bound, given next, is to define a set of *generic requests* (*i.e.*, virtual requests defined for analysis purposes) that upper-bound the worst-case contention during any interval of length $t_1 - t_0$. That is, the interference bound for an interval of length $t_1 - t_0$ contains at least as many requests as T_i issues in any interval of length $t_1 - t_0$ in any actual schedule, and each generic request is at least as long as a corresponding actual one. This can be formalized as follows.

Definition 7 The *task interference bound* for an interval of length t, denoted $tif(T_i, \ell_q, t)$, is a set of generic requests that satisfies the following two properties.

1. For any $C_{i,q}(t_0, t_1)$ with regard to some actual schedule, one can choose a set of corresponding generic requests $C'_{i,q} \subseteq tif(T_i, \ell_q, t_1 - t_0)$ that satisfies

$$\left|C'_{i,q}\right| = |C_{i,q}(t_0, t_1)| \qquad \text{and} \qquad \sum_{\mathcal{R}_{i,q,v} \in C_{i,q}(t_0,t_1)} \mathcal{L}_{i,q,v} \quad \leq \sum_{\mathcal{R}_{i,q,w} \in C'_{i,q}} \mathcal{L}_{i,q,w}.$$

2. Interference bounds are inclusive:

$$t \leq t' \Rightarrow tif(T_i, \ell_q, t) \subseteq tif(T_i, \ell_q, t').$$

Property 1 ensures that the task interference bound does not underestimate the number and length of requests in any actual execution of T_i, and Property 2 ensures that a derived bound remains valid when analyzing a larger-than-necessary interval (*i.e.*, when over-estimating a job's response time). In the case of RW constraints, we analogously define task T_i's *read interference bound*, denoted as $rif(T_i, \ell_q, t)$, with respect to read requests for ℓ_q, and T_i's *write interference bound*, denoted as $wif(T_i, \ell_q, t)$ with respect to write requests for ℓ_q.

These definitions serve as an interface that allows the analysis of specific lock types presented in the following sections to be seamlessly integrated with more-refined task and resource models. Next, we provide a suitable definition of $tif(T_i, \ell_q, t)$ for the model assumed in this paper. To this end, we require the following well-known bound on the maximum number of jobs that can execute requests in a given interval. Recall from Section 2 that p_i denotes T_i's period and r_i denotes T_i's maximum response time.

Lemma 16 *At most* $\left\lceil \dfrac{t + r_i}{p_i} \right\rceil$ *distinct jobs of a task* T_i *can execute in any interval of length* t *(without proof, see e.g. [14, 18]).*

It follows from Lemma 16 and the definition of $N_{i,q}$ that jobs of T_i issue at most $\lceil (t + r_i)/p_i \rceil \cdot N_{i,q}$ requests for ℓ_q over any interval of length t. In the worst case, each request for ℓ_q is of length $L_{i,q}$. This yields the following interference bound for the task model assumed herein.

Definition 8 The *request interference bound* for task T_i with respect to resource ℓ_q over any interval of length t is the set of requests

$$tif(T_i, \ell_q, t) \triangleq \left\{ \mathcal{R}_{i,q,v} \mid 1 \leq v \leq N_{i,q} \cdot \left\lceil \frac{t + r_i}{p_i} \right\rceil \right\},$$

where $\mathcal{L}_{i,q,v} = L_{i,q}$ for each $\mathcal{R}_{i,q,v}$. If T_i does not access a given resource ℓ_q, then $tif(T_i, \ell_q, t) = \emptyset$ for all t. We analogously define task T_i's read and write interference bounds as $rif(T_i, \ell_q, t)$ and $wif(T_i, \ell_q, t)$, respectively.

Based on per-task interference bounds, we next introduce a generic, parametrized "aggregate interference bound" for use in the subsequent analysis of specific locking protocols. We first define three convenience functions over sets of requests, which serve to simplify the expression of "aggregate interference" and protocol-specific bounds on blocking.

Definition 9 Given a set of requests S, we let S_k denote the k^{th} longest request in S, where $1 \leq k \leq |S|$ (with ties broken arbitrarily but consistently). Formally, if $1 \leq k \leq l \leq |S|$ and $S_k = \mathcal{R}_{a,b,c}$ and $S_l = \mathcal{R}_{x,y,z}$, then $\mathcal{L}_{a,b,c} \geq \mathcal{L}_{x,y,z}$.

Definition 10 Given a set of requests S, we denote the set of the l longest requests in S as $top(l, S) \triangleq \{S_k \mid 1 \leq k \leq \min(l, |S|)\}$ and their total duration as $total(l, S) \triangleq \sum_{\mathcal{R}_{i,q,v} \in top(l,S)} \mathcal{L}_{i,q,v}$. If $l = 0$ or $S = \emptyset$, then $total(l, S) = 0$.

A task interference bound limits the maximum contention from jobs of a single task. Using the above definitions, we can formalize the notion of contention from a set of tasks. Recall Constraint 3 from Section A.1 above, namely that the number of requests per task that can possibly cause J_i to incur acquisition delay is limited if jobs wait in FIFO order. If a task T_x can delay J_i with at most l requests, then it is sufficient to consider only the l longest requests in T_x's interference bound. We therefore define the aggregate interference bound with a per-task "interference limit" parameter.

Definition 11 The *aggregate interference bound* of a set of tasks τ with respect to a resource ℓ_q over any interval of length t and subject to an *interference limit* l is given by

$$tifs(\tau, \ell_q, t, l) \triangleq \bigcup_{T_x \in \tau} top\left(l,\ tif(T_x, \ell_q, t)\right).$$

A task set's *aggregate read interference* and *aggregate write interference*, denoted as $rifs(\tau, \ell_q, t, l)$ and $wifs(\tau, \ell_q, t, l)$, respectively, are defined analogously with respect to read and write interference.

Given an interference limit l, $tifs(\tau, \ell_q, t, l)$ contains the l longest requests in each task's interference bound for ℓ_q and t. In the task model assumed in this paper, each request in $tif(T_x, \ell_q, t)$ is in fact of the same length $\mathcal{L}_{x,q,v} = L_{x,q}$ (see Definition 8). We define $tifs(\tau, \ell_q, t, l)$ with additional generality to accomodate more-expressive task models for which $tif(T_x, \ell_q, t)$ may contain non-uniform request lengths.

The holistic blocking analysis framework incorporates Constraints 1 and 3 from Section A.1 in a generic fashion. The remaining Constraints 2 and 4 are easier to incorporate on a protocol-by-protocol basis, which we do next to derive concrete, non-asymptotic bounds for the locking protocols presented in this paper.

A.3 The Global OMLP for Mutual Exclusion

We begin with the global OMLP for mutex constraints under s-oblivious schedulability analysis. Since the global OMLP uses a hybrid queue that consists of a FIFO queue FQ_q (which holds at most m jobs) and of a priority queue PQ_q (which is only used if at least $m + 1$ jobs are queued), maximum s-oblivious pi-blocking under the global OMLP depends on how many tasks share a given resource.

Definition 12 In the following, let $A_q \triangleq |\{T_i \mid T_i \in \tau \wedge N_{i,q} > 0\}|$ denote the number of tasks that access resource ℓ_q.

If $A_q \leq m + 1$, then at most m jobs are waiting to acquire ℓ_q at any time, which implies that at most one job is queued in PQ_q. In this case, the global OMLP reduces to a simple FIFO protocol.

Lemma 17 *Under the global OMLP, if $A_q \leq m + 1$, then a job J_i incurs at most*

$$b_{i,q} = total((A_q - 1) \cdot N_{i,q},\ tifs(\tau \setminus \{T_i\}, \ell_q, r_i, N_{i,q}))$$

s-oblivious pi-blocking due to requests for resource ℓ_q.

Proof J_i's response time r_i upper-bounds the duration of the interval during which other jobs can issue conflicting requests; that is, the aggregate task interference bound $tifs(\tau \setminus \{T_i\}, r_i, N_{i,q})$ for any interval of length r_i is a sufficient approximation of the resource demands of competing tasks. If J_i is never enqueued in PQ_q, then the lemma follows trivially.

Otherwise, if J_i is enqueued in PQ_q, then m jobs are already enqueued in FQ_q at the time of J_i's request. Since $A_q \leq m + 1$, this implies that no other job is enqueued in PQ_q. As soon as the head of FQ_q releases ℓ_q, J_i is moved to FQ_q. Hence there is at most one job in PQ_q at any time, and the ordering of PQ_q is irrelevant.

The FIFO ordering of FQ_q implies that each of J_i's requests is preceded by at most one request from each other task that accesses ℓ_q. The per-task interference limit is hence $N_{i,q}$. Since ℓ_q is shared among only $A_q \leq m + 1$ tasks, one of which is T_i, no more than $(A_q - 1) \cdot N_{i,q}$ requests pi-block J_i in total. Priority inheritance ensures that the resource-holding job is scheduled whenever J_i incurs s-oblivious pi-blocking; the cumulative duration of the $(A_q - 1) \cdot N_{i,q}$ longest requests for ℓ_q by tasks other than J_i thus bounds maximum s-oblivious pi-blocking. □

In the case of $A_q > m + 1$, higher-priority jobs of some other task T_x may "skip ahead" of J_i repeatedly while J_i waits in PQ_q. However, the per-task interference limit is still limited to $2 \cdot N_{i,q}$, that is, the per-task interference limit is only doubled even if jobs of T_x "skip ahead" an arbitrary number of times.

Lemma 18 *Let T_x denote some task other than T_i that accesses ℓ_q (i.e., $T_i \neq T_x$ and $N_{x,q} > 1$). Under the global OMLP, jobs of T_x cause J_i to incur s-oblivious pi-blocking for at most the duration of two requests each time that J_i requests ℓ_q.*

Proof In order to pi-block J_i, a request issued by some J_x must precede J_i's request in FQ_q (*i.e.*, J_x enters FQ_q before J_i does). If $A_q \leq m+1$, the bound follows analogously to Lemma 17 since FQ_q is FIFO-ordered.

Hence assume $A_q > m+1$. In this case, jobs of T_x may enter FQ_q repeatedly while J_i waits in PQ_q. Let t_a denote the first time that a job of T_x, denoted $J_{x,a}$, enters FQ_q, and let t_b denote the second time that a job of T_x, denoted $J_{x,b}$, enters FQ_q while J_i is continuously waiting in PQ_q. Since tasks are sequential, $J_{x,b}$ necessarily issued its request after J_i issued its request (this is not necessarily the case with $J_{x,a}$).

Further, let t_1 denote the time that J_i enters FQ_q (as indicated in Figure 10). If t_1 does not exist (*i.e.*, if J_i never enters FQ_q), then either FQ_q is continuously populated with higher-priority jobs and J_i does not incur s-oblivious pi-blocking, or some requests fails to complete (which is not possible since each $L_{i,q}$ is presumed finite). Therefore assume t_1 exists.

J_i does not incur s-oblivious pi-blocking during $[t_b, t_1)$. Since J_i is waiting in PQ_q at time t_a, $J_{x,a}$ is necessarily preceded by $m-1$ other jobs in FQ_q, which must complete before $J_{x,a}$'s request is satisfied. Since tasks are sequential, $J_{x,a}$ has completed its request before $J_{x,b}$ enters FQ_q at time t_b. Therefore, at least m higher-priority jobs must have entered FQ_q during $[t_a, t_b)$; otherwise, J_i would no longer be waiting in PQ_q at time t_b. The presence of m higher-priority pending jobs rules out s-oblivious pi-blocking after t_b (until J_i enters FQ_q at time t_1).

Therefore, at most one of the requests issued by jobs of T_x *after* J_i issued its request pi-blocks J_i. Since sporadic tasks are sequential, at most one request of T_x that was issued *prior* to J_i's request is incomplete when J_i issues its request. Hence, at most two requests of T_x cause J_i to incur pi-blocking. □

As a result, the per-task interference limit in the case of $A_q > m+1$ is $2 \cdot N_{i,q}$. This yields the following bound.

Lemma 19 *Under the global OMLP, if $A_q > m+1$, then a job J_i incurs at most*

$$b_{i,q} = total((2 \cdot m - 1) \cdot N_{i,q},\ tifs(\tau \setminus \{T_i\}, \ell_q, r_i, 2 \cdot N_{i,q}))$$

s-oblivious pi-blocking due to requests for resource ℓ_q.

Proof By Lemma 14, J_i incurs s-oblivious pi-blocking for the combined duration of at most $2 \cdot m - 1$ requests each time that it requests ℓ_q, which implies that J_i is delayed by at most $(2 \cdot m - 1) \cdot N_{i,q}$ requests in total. Lemma 18 implies an interference limit of $2 \cdot N_{i,q}$. Priority inheritance ensures that the resource-holding job is scheduled whenever J_i incurs pi-blocking. The bound follows. □

This yields the following overall bound on maximum s-oblivious pi-blocking.

Theorem 4 *Under the global OMLP, a job J_i incurs s-oblivious pi-blocking for at most*

$$b_i = \sum_{q=1}^{n_r} total((x_q - 1) \cdot N_{i,q},\ tifs(\tau \setminus \{T_i\}, \ell_q, r_i, l_q \cdot N_{i,q}))$$

time units, where $x_q \triangleq A_q$ and $l_q \triangleq 1$ if $A_q \leq m+1$, and $x_q \triangleq 2 \cdot m$ and $l_q \triangleq 2$ if $A_q > m+1$.

Proof Follows from Lemmas 17 and 19, since resource requests are not nested, and since J_i does not incur s-oblivious pi-blocking under the global OMLP while not requesting resources. □

This concludes the analysis of the global OMLP. Next, we consider the clustered OMLP from Sections 4.2–4.4, which uses priority donation instead of priority inheritance.

A.4 The Clustered OMLP for Mutual Exclusion

A job J_i is subject to two sources of s-oblivious pi-blocking under the clustered OMLP. J_i can be delayed each time it issues requests for shared resources, and additionally once upon release if it serves as a priority donor. We begin with the mutex variant of the clustered OMLP, which is the simplest of the three protocols based on priority donation. Recall from Section 4.2 that each resource ℓ_q is protected by a simple FIFO queue FQ_q.

Lemma 20 *Under the clustered OMLP's mutex protocol, a job J_i incurs at most*

$$b_{i,q,j} = \begin{cases} total(N_{i,q} \cdot c,\ tifs(\tau_j, \ell_q, r_i, N_{i,q})) & \text{if } j \neq P_i \\ total(N_{i,q} \cdot (c-1),\ tifs(\tau_j \setminus \{T_i\}, \ell_q, r_i, N_{i,q})) & \text{if } j = P_i \end{cases}$$

pi-blocking due to requests for resource ℓ_q issued by jobs of tasks assigned to the j^{th} cluster.

Proof By Lemma 4, priority donation ensures that at most c requests are incomplete at any time in each cluster; therefore, at most c requests from each cluster C_j precede J_i in FQ_q each time that it issues a request. The strict FIFO ordering in FQ_q ensures a per-task interference limit of $N_{i,q}$. Due to priority donation, resource-holding jobs are always scheduled (Lemma 2). In the case of J_i's local cluster (*i.e.*, if $j = P_i$), only $c - 1$ requests can interfere since J_i's own request counts towards the limit of c concurrent requests imposed by priority donation. Since jobs and tasks are sequential, J_i is not delayed by requests of (other) jobs of T_i. □

When bounding the maximum pi-blocking due to priority donation, we only need to consider the set of tasks that could have released a lower-priority job prior to J_i's arrival since priorities are only donated to jobs with lower base priority. This set of tasks necessarily depends on the specific scheduling policy.

Definition 13 We let $lower(T_i)$ denote the set of local tasks that could potentially require one of T_i's jobs to serve as a priority donor upon release. Under EDF-based schedulers, $lower(T_i)$ includes only tasks with longer relative deadlines. Under FP-based schedulers, $lower(T_i)$ includes tasks with lower priorities.

Lemma 21 *Under the clustered OMLP's mutex protocol, a job J_i incurs at most b_i^D s-oblivious pi-blocking upon release while serving as a priority donor, where*

$$b_i^D = \max_{1 \le q \le n_r} \max_{\substack{T_x \in lower(T_i) \\ N_{x,q} > 0}} \left(L_{x,q} + \sum_{j=1}^{m/c} b'_{x,q,j} \right), \quad \textit{and}$$

$$b'_{x,q,j} = \begin{cases} total(c,\ tifs(\tau_j, \ell_q, r_x, 1)) & \textit{if } j \neq P_x, \\ total(c-1,\ tifs(\tau_j \setminus \{T_i, T_x\}, \ell_q, r_x, 1)) & \textit{if } j = P_x. \end{cases}$$

Proof By Lemma 3, maximum s-oblivious pi-blocking due to priority donation is limited to one request span. Analogously to Lemma 20, b_i^D bounds the maximum request span of any local, potentially lower-priority job J_x by considering the c longest requests in each remote cluster that could cause J_x to incur acquisition delay, and the $c - 1$ longest requests in J_x's local cluster. □

Theorem 5 *Under the clustered OMLP's mutex protocol, a job J_i incurs at most*

$$b_i = b_i^D + \sum_{q=1}^{n_r} \sum_{j=1}^{m/c} b_{i,q,j}$$

s-oblivious pi-blocking due to requests for shared resources, where $b_{i,q,j}$ and b_i^D are defined as in Lemmas 20 and 21, respectively.

Proof Follows from Lemmas 20 and 21, and the assumptions that resource requests are not nested and that tasks do not migrate across cluster boundaries. □

A.5 The Clustered OMLP for RW Exclusion

The bounds on maximum pi-blocking under the OMLP's RW protocol are structurally similar to the bounds on maximum spin-blocking and pi-blocking under non-preemptive phase-fair RW spinlocks that we previously presented in [18]. This is because the OMLP implements phase-fairness, and because priority donation allows at most c concurrent requests in each cluster, which has an effect that is equivalent to non-preemptive execution.

We begin by considering the set of potentially blocking write requests. Since write requests are satisfied in FIFO order with respect to other write requests, maximum acquisition delay *incurred by a writer* due to earlier-issued write requests is the same under the OMLP's mutex and RW variants. However, since reader and writer phases alternate, the maximum acquisition delay *incurred by a reader* due to earlier-issued write requests is limited to one critical section. That is, at most $N_{i,q}^W \cdot c + N_{i,q}^R$ write requests issued by jobs of a remote cluster can block J_i under the clustered OMLP's RW variant. In the case of J_i's local cluster, if $c > 1$, then the same reasoning applies and no more than $N_{i,q}^W \cdot (c-1) + N_{i,q}^R$ write requests block J_i. In the special case of $c = 1$, local jobs cannot cause J_i to incur acquisition delay since they are not scheduled while J_i waits. These considerations lead to the following definition of the set of possibly-interfering write requests.

Definition 14 In the following, let $x^{rem} = N_{i,q}^W \cdot c + N_{i,q}^R$ and $x^{loc} = N_{i,q}^W \cdot (c-1) + N_{i,q}^R$, and define the sets of possibly-interfering write requests from jobs in the j^{th} cluster, denoted as $W(T_i, j, \ell_q)$, as follows.

$$W(T_i, j, \ell_q) = \begin{cases} top(x^{rem},\ wifs(\tau_j, \ell_q, r_i, (N_{i,q}^W + N_{i,q}^R))) & \text{if } j \neq P_i \\ top(x^{loc},\ wifs(\tau_j \setminus \{T_i\}, \ell_q, r_i, (N_{i,q}^W + N_{i,q}^R))) & \text{if } j = P_i \text{ and } c > 1 \\ \emptyset & \text{if } j = P_i \text{ and } c = 1 \end{cases}$$

Further, let $W_{i,q}$ denote the union of all possibly-interfering write requests across all clusters, and let $w_{i,q}$ denote the maximum number of blocking write requests.

$$W_{i,q} = \bigcup_{j=1}^{m/c} W(T_i, j, \ell_q) \qquad\qquad w_{i,q} = |W_{i,q}|$$

Next, we consider the set of potentially blocking read requests. The defining property of an RW lock is that readers do not *directly* block other readers. That is, in the absence of any writers, a reader is not delayed in RW locks regardless of the number of concurrent read requests. Intuitively, a reader phase can only *transitively* block another read request if said phase is "assisted" by an also-blocking, interspersed write request. This intuition can be formalized to characterize acquisition delay due to interfering read requests in terms of the number of interfering write requests.

Lemma 22 (from [14, 18]) *Let J_i denote a job that issues at most $N_{i,q}^W$ write requests for a resource ℓ_q, let w denote the number of write requests that cause J_i's write requests for ℓ_q to incur acquisition delay, and let r denote the number of reader phases that cause J_i's write requests for ℓ_q to incur acquisition delay. If ℓ_q is protected by a phase-fair RW lock, then $r \leq w + N_{i,q}^W$.*

Similarly, a writer that is not delayed by other writers incurs acquisition delay for the duration of at most one read request regardless of the number of blocking readers. For example, if $m-1$ readers hold a resource ℓ_q when J_i issues a write request for ℓ_q, then all $m-1$ readers proceed in parallel and J_i incurs acquisition delay only for the duration of the longest earlier-issued read request. Therefore, J_i incurs acquisition delay due to interfering read requests for the combined duration of at most $N_{i,q}^R + (m-1) \cdot N_{i,q}^W$ read requests (recall Lemmas 8 and 9). Taken together, this leads to the following definition.

Definition 15 Let $r_{i,q} = \min(w_{i,q} + N_{i,q}^W,\ N_{i,q}^R + (m-1) \cdot N_{i,q}^W)$, and define the sets of possibly-interfering read requests from jobs in the j^{th} cluster, denoted as $R(T_i, j, \ell_q)$, as follows.

$$R(T_i, j, \ell_q) = \begin{cases} top(r_{i,q},\ rifs(\tau_j, \ell_q, r_i, r_{i,q})) & \text{if } j \neq P_i \\ top(r_{i,q},\ rifs(\tau_j \setminus \{T_i\}, \ell_q, r_i, r_{i,q})) & \text{if } j = P_i \text{ and } c > 1 \\ \emptyset & \text{if } j = P_i \text{ and } c = 1 \end{cases}$$

Analogously to $W_{i,q}$, let $R_{i,q}$ denote the set of all possibly interfering read requests across all clusters.

$$R_{i,q} = \bigcup_{j=1}^{m/c} R(T_i, j, \ell_q)$$

With these definitions in place, we can state the following bound on pi-blocking due to requests for a given resource.

Lemma 23 *Under the clustered OMLP's RW protocol, a job J_i incurs pi-blocking due to its read and write requests for resource ℓ_q for at most $b_{i,q} = total(w_{i,q},\ W_{i,q}) + total(r_{i,q},\ R_{i,q})$ time units.*

Proof Analogously to Lemma 20. Each time that J_i issues a write request, it can be preceded by up to c other write requests in each cluster since the writer queue WQ_q is FIFO ordered, and because priority donation allows at most c concurrent requests per cluster. Also due to the FIFO order, each other task can block each of J_i's write requests with at most one request. Each time that J_i issues a read request, it is blocked by at most one write request since the OMLP implements phase-fairness. Therefore, the per-task interference with regard to write requests is $N_{i,q}^W + N_{i,q}^R$, and in total J_i's $N_{i,q}^R$ read requests and $N_{i,q}^W$ write requests are blocked by at most $N_{i,q}^R + N_{i,q}^W \cdot c$ write requests in the case of a remote cluster, and by at most $N_{i,q}^R + N_{i,q}^W \cdot (c-1)$ requests in the case of J_i's local cluster. The definitions of $W(T_i, j, \ell_{i,q})$ and $W_{i,q}$ follow.

By Lemma 22, the upper bound on the total number of blocking writes $w_{i,q}$ implies an upper bound of $w_{i,q} + N_{i,q}^W$ on the number of blocking reader phases. The total number of blocking reader phases is also limited to $N_{i,q}^R + (m-1) \cdot N_{i,q}^W$: due to priority donation and because reader and writer phases alternate in a phase-fair RW lock, each of J_i's read requests is transitively blocked by at most one reader phase, and each of J_i's write requests is blocked by at most $m-1$ interspersed reader phases (since at most $m-1$ write requests block each of J_i's write requests). The lesser of the two bounds limits the total number of blocking reader phases $r_{i,q}$. The definitions of $R(T_i, j, \ell_q)$ and $R_{i,q}$ follow.

Since J_i is blocked by at most $w_{i,q}$ writer phases and $r_{i,q}$ reader phases, total s-oblivious pi-blocking is bounded by the $w_{i,q}$ longest requests in $W_{i,q}$ and the $r_{i,q}$ longest request in $R_{i,q}$. □

Since the clustered OMLP uses priority donation, a job may also incur s-oblivious pi-blocking when serving as a priority donor. The duration of priority donation depends on the request span of the priority recipient's request, which may be either a write or a read. The maximum acquisition delay of a single write request for resource ℓ_q issued by job J_i can be bounded by instantiating Definitions 14 and 15 assuming $N_{i,q}^R = 0$ and $N_{i,q}^W = 1$. Similarly, the maximum acquisition delay of a single read request for ℓ_q can be bounded by instantiating said definitions assuming $N_{i,q}^R = 1$ and $N_{i,q}^W = 0$. To avoid needless repetition, we use the following definitions to denote these two special cases.

Definition 16 Let $W'_{i,q}$ and $w'_{i,q}$ denote the values of $W_{i,q}$ and $w_{i,q}$, respectively, that result when assuming $N^R_{i,q} = 0$ and $N^W_{i,q} = 1$ in Definition 14 above. Similarly, let $W''_{i,q}$ and $w''_{i,q}$ denote the values of $W_{i,q}$ and $w_{i,q}$, respectively, that result when assuming $N^R_{i,q} = 1$ and $N^W_{i,q} = 0$ in Definition 14 above.

Definition 17 Let $R'_{i,q}$ and $r'_{i,q}$ denote the values of $R_{i,q}$ and $r_{i,q}$, respectively, that result when assuming $N^R_{i,q} = 0$ and $N^W_{i,q} = 1$ in Definition 15 above. Similarly, let $R''_{i,q}$ and $R''_{i,q}$ denote the values of $R_{i,q}$ and $R_{i,q}$, respectively, that result when assuming $N^R_{i,q} = 1$ and $N^W_{i,q} = 0$ in Definition 15 above.

With these special cases in place, we can express the maximum request span. Recall from Definition 13 that we let $lower(T_i)$ denote the set of tasks local to T_i that could potentially cause J_i to incur pi-blocking upon release.

Lemma 24 *Under the clustered OMLP's RW protocol, a job J_i incurs at most $b^D_i = \max(b'_i, b''_i)$ s-oblivious pi-blocking upon release while serving as a priority donor, where*

$$b'_i = \max_{1 \le q \le n_r} \left\{ L^W_{x,q} + b'_{x,q} \right\}, \text{ and } \max_{\substack{T_x \in lower(T_i) \\ N^W_{x,q} > 0}}$$

$$b'_{x,q} = total(w'_{x,q},\ W'_{x,q}) + total(r'_{x,q},\ R'_{x,q}),$$

bounds the case of a writing priority recipient, and where

$$b''_i = \max_{1 \le q \le n_r} \left\{ L^R_{x,q} + b''_{x,q} \right\}, \text{ and } \max_{\substack{T_x \in lower(T_i) \\ N^R_{x,q} > 0}}$$

$$b''_{x,q} = total(w''_{x,q},\ W''_{x,q}) + total(r''_{x,q},\ R''_{x,q}).$$

bounds the case of a reading priority recipient.

Proof Follows analogously to Lemma 21 since J_i serves as a priority donor at most once and at most for the duration of one request span. The maximum request span of a lower-priority write request is bounded by b'_i; the maximum request span of a lower-priority read request is bounded by b''_i. The maximum of either scenario bounds maximum s-oblivious pi-blocking due to priority donation under the clustered OMLP for RW exclusion. □

This yields the following bound on s-oblivious pi-blocking.

Theorem 6 *Under the clustered OMLP's mutex protocol, a job J_i incurs at most*

$$b_i = b^D_i + \sum_{q=1}^{n_r} b_{i,q}$$

s-oblivious pi-blocking due to read and write requests for shared resources, where $b_{i,q}$ and b^D_i are defined as in Lemmas 23 and 24, respectively.

Proof Follows from Lemmas 23 and 24, and the assumptions that resource requests are not nested and that tasks do not migrate across cluster boundaries. □

A.6 The Clustered OMLP for k-Exclusion

In this section, we establish a bound on s-oblivious pi-blocking under the clustered OMLP for k-exclusion, which is presented in Section 4.4. The presented analysis is reasonably tight if blocking requests are relatively uniform in duration. However, if request lengths are heavily skewed (*i.e.*, if there are some infrequent, long-running requests, but most requests are short), then a more accurate bound could be obtained by applying multiprocessor response-time analysis for non-preemptive global FIFO scheduling to each resource. In the following simpler analysis, which suffices for our purposes, some pessimism arises because Lemma 11, which implicitly lower-bounds the request completion rate, does not take non-uniform request lengths into account.

Lemma 25 *Under the clustered OMLP's k-exclusion protocol, a job J_i incurs at most $b_{i,q}$ s-oblivious pi-blocking due to requests for resource ℓ_q, where*

$$b_{i,q} = total\left(N_{i,q} \cdot \left\lceil \frac{m - k_q}{k_q} \right\rceil,\ \bigcup_{j=1}^{m/c} b_{i,q,j} \right), \quad \text{and}$$

$$b_{i,q,j} = \begin{cases} top(N_{i,q} \cdot c,\ tifs(\tau_j, \ell_q, r_i, N_{i,q})) & \text{if } j \neq P_i \\ top(N_{i,q} \cdot (c-1),\ tifs(\tau_j \setminus \{T_i\}, \ell_q, r_i, N_{i,q})) & \text{if } j = P_i. \end{cases}$$

Proof By Lemma 4, priority donation ensures that at most c requests are incomplete at any time in each cluster; therefore, at most c requests in each cluster precede J_i in KQ_q or hold a replica of ℓ_q at the time that J_i issues a request. The FIFO ordering of jobs in KQ_q ensures a per-task interference limit of $N_{i,q}$. Therefore, the set of the $N_{i,q} \cdot c$ longest requests issued by jobs in the j^{th} cluster, denoted $b_{i,q,j}$, bounds the worst-case interference from jobs in that cluster. In the case of J_i's local cluster, only $c-1$ requests can interfere since J_i's own request counts towards the limit of c concurrent requests imposed by priority donation.

Lemma 11 implies that J_i holds a replica of ℓ_q after at most $\lceil (m-k_q)/k_q \rceil$ prior requests for ℓ_q complete. Therefore, across all $N_{i,q}$ requests, J_i is pi-blocked at most for the cumulative duration of the $N_{i,q} \cdot \lceil (m-k_q)/k_q \rceil$ longest requests issued by jobs in any cluster. □

To bound maximum s-oblivious pi-blocking due to priority donation, we again require a bound for a single request. Such a bound can be obtained by applying Lemma 25 above to a single request.

Definition 18 Let $b'_{i,q}$ denote the value of $b_{i,q}$ computed assuming $N_{i,q} = 1$ in Lemma 25 above.

Recall from Definition 13 that we let $lower(T_i)$ denote the set of tasks local to T_i that could potentially cause J_i to incur pi-blocking upon release.

Lemma 26 *Under the clustered OMLP's k-exclusion protocol, a job J_i incurs at most*

$$b_i^D = \max_{1 \leq q \leq n_r} \left\{ \max_{\substack{T_x \in lower(T_i) \\ N_{x,q} > 0}} \{L_{x,q} + b'_{x,q}\} \right\}$$

s-oblivious pi-blocking upon release while serving as a priority donor.

Proof Follows analogously to Lemma 21 and Lemma 24. □

Theorem 7 *Under the clustered OMLP's k-exclusion protocol, a job J_i incurs at most*

$$b_i = b_i^D + \sum_{q=1}^{n_r} b_{i,q}$$

s-oblivious pi-blocking due to requests for shared resources, where $b_{i,q}$ and b_i^D are defined as in Lemmas 25 and 26, respectively.

Proof Follows from Lemmas 25 and 26, and since resource requests are not nested. □

A.7 Schedulability Test

Having derived bounds on maximum s-oblivious pi-blocking, any *sustainable* [6, 8] locking-unaware schedulability test can be used to establish schedulability under the OMLP. In short, we require a sustainable schedulability test because each task's parameter b_i is only an uppper bound (*i.e.*, it is not exact); therefore the employed schedulability test must be resilient to execution cost decreases at runtime.

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 schedulability tests that assume tasks to be independent.

Theorem 8 *Let $\mathcal{T}$ denote a sustainable schedulability test for independent tasks for the employed JLFP scheduling policy. A task set τ is schedulable under the OMLP if $\tau' \triangleq \left\{ T'_i(e_i + b_i, p_i) \mid T_i \in \tau \right\}$ is deemed schedulable by $\mathcal{T}$.*

Note that the derivation of b_i itself does not depend on the actual scheduling policy or $\mathcal{T}$; the OMLP can thus be applied to any JLFP scheduling policy and any corresponding sustainable schedulability test.