Real-Time Resource-Sharing under Clustered Scheduling: Mutex, Reader-Writer, and k-Exclusion Locks*

Björn B. Brandenburg
Department of Computer Science
University of North Carolina at Chapel Hill
bbb@cs.unc.edu

James H. Anderson
Department of Computer Science
University of North Carolina at Chapel Hill
anderson@cs.unc.edu

ABSTRACT

This paper presents the first suspension-based real-time locking protocols for clustered schedulers. Such schedulers pose challenges from a locking perspective because they exhibit aspects of both partitioned and global scheduling, which seem to necessitate fundamentally different means for bounding priority inversions. A new mechanism to bound such inversions, termed *priority donation*, is presented and used to derive protocols for mutual exclusion, reader-writer exclusion, and k-exclusion. Each protocol has asymptotically optimal blocking bounds under certain analysis assumptions. The latter two protocols are also the first of their kind for the special cases of global and partitioned scheduling.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-Time and Embedded Systems*; D.4.1 [**Operating Systems**]: Process Management—*Multiprocessing; Mutual Exclusion; Scheduling; Synchronization*

General Terms

Algorithms, Performance, Verification

1. INTRODUCTION

Recent experimental work has demonstrated the effectiveness of *clustered* scheduling on large multicore, multi-chip platforms [4]. Clustered scheduling [2, 11] is a generalization of both *partitioned* scheduling (one ready queue per processor) and *global* scheduling (all processors serve a single ready queue), where tasks are partitioned onto clusters of cores and a global scheduling policy is used within each cluster. 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 is an attractive compromise between these two extremes because it both simplifies the task assignment problem (there are fewer and larger bins) and incurs less overhead (by aligning clusters with the underlying hardware topology). Consequently, clustered scheduling is likely to grow in importance as multicore platforms become larger and less uniform.

To be practical, a scheduler must support locking protocols that allow tasks predictable access to shared resources such as I/O devices. There are two approaches to implementing such protocols: in *spin-based* protocols, jobs wait for resources by executing a delay loop, and in *suspension-based* protocols, waiting jobs relinquish their processor. In this paper, we focus on suspension-based protocols.

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*, *i.e.*, if tasks use resources for at most a few microseconds [6, 10].

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. Unfortunately, no suspension-based real-time locking protocols have been proposed for clustered scheduling to date. Worse, the established mechanisms for bounding priority inversions do not transfer to clustered scheduling.

Priority inversion. The main goal in the design of real-time locking protocols is to minimize the duration of priority inversions, which (intuitively) occur when a high-priority job must wait for a lower-priority one. Under global scheduling, this is commonly achieved using priority inheritance, whereas priority boosting is employed under partitioning (see Sec. 2 for definitions). However, as shown later, neither mechanism works under clustered scheduling: priority inheritance is ineffective across cluster boundaries and priority boosting allows high-priority jobs to be blocked repeatedly.

In this paper, we tackle this troublesome situation by developing a new mechanism to bound priority inversions—termed "priority donation"—that causes jobs to be blocked at most once. Based on "priority donation," we design novel suspension-based locking protocols that work under any clustered job-level static-priority (JLSP) scheduler for three common resource-sharing constraints: **(i)** *mutual exclusion* (mutex), where every resource access must be exclusive; **(ii)** *reader-writer* (RW) exclusion, where only updates must be exclusive and reads may overlap with each other; and **(iii)** k-*exclusion*, where there are k replicas of a resource and tasks require exclusive access to any one replica.

Related work. Most prior work has been directed at *earliest-deadline-first* (EDF) and *static-priority* (SP) scheduling, which are both JLSP policies, as well as at their partitioned and global multiprocessor extensions (denoted as PSP, PEDF, GSP, and GEDF, resp.). The classic uniprocessor *stack resource policy* (SRP) [1] and

*Work supported by 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.

EMSOFT'11, October 9–14, 2011, Taipei, Taiwan.

the *priority ceiling protocol* (PCP) [19, 21] both support *multi-unit resources*, which is a generalized resource model that can be used to realize mutex, RW, and k-exclusion constraints.

Work on multiprocessor protocols has mostly focused on mutex constraints to date. The first such protocols were proposed by Rajkumar *et al.* [18, 19, 20], who designed two suspension-based PCP extensions for PSP-scheduled systems, the *distributed* and the *multiprocessor priority ceiling protocol* (DPCP and MPCP, resp.), which augment priority inheritance with priority boosting. In later work on PEDF-scheduled systems, suspension- and spin-based protocols were presented by Chen and Tripathi [13] and Gai *et al.* [16]. Block *et al.* [5] recently presented the *flexible multiprocessor locking protocol* (FMLP), which can be used under GEDF, PEDF, and PSP [7] and supports both spin- and suspension-based waiting. More recently, Easwaran and Andersson [14] considered suspension-based protocols for GSP-scheduled systems. Finally, Faggioli *et al.* [15] presented a scheduler-agnostic spin-based protocol for mixed real-time/non-real-time environments.

In [9], 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 blocking, as an alternative.

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, we are not aware of relevant analysis.

Pi-blocking. In other recent work [8], we investigated asymptotic bounds on priority-inversion blocking (*pi-blocking*) in the context of mutex constraints. We found that the definition of pi-blocking is actually analysis-dependent, as there are two approaches to handling task suspensions (which are notoriously hard to analyze). The first approach, *suspension-oblivious* schedulability analysis, does not allow for suspension times to be explicitly accounted for. This lack of expressivity in the task model necessitates such times to be modeled as computation instead. Consequently, suspension-oblivious analysis over-estimates the processor demand of resource-sharing tasks and thereby yields pessimistic but sound results. In contrast, *suspension-aware* schedulability analysis, considers suspensions explicitly and thus uses less-pessimistic processor demand estimates.

For suspension-oblivious analysis, we established a lower bound of $\Omega(m)$ on pi-blocking (per resource request) for any m-processor locking protocol (under any JLSP scheduler). We also devised a new mutex protocol for global and partitioned scheduling, the *$O(m)$ locking protocol* (OMLP), that has $O(m)$ suspension-oblivious pi-blocking and is thus asymptotically optimal. Perhaps surprisingly, the improvement in analysis accuracy in suspension-aware analysis comes at the cost of an *increased* lower bound for mutex protocols: we established a lower bound of $\Omega(n)$ on pi-blocking, where n is the number of tasks in the system. The difference in lower bounds arises because the nature of what constitutes a "priority inversion" is changed by the assumption underlying suspension-oblivious analysis. Intuitively, the analytical trick is to "reuse" some of the pessimism inherent in treating suspensions as execution time to derive less pessimistic bounds on priority inversion length.

Contributions. In this paper, we consider locking protocols for clustered JLSP schedulers. We focus on the suspension-oblivious case because virtually all current global scheduling analysis results (which are needed to analyze each cluster) are suspension-oblivious (this restriction is revisited in Sec. 4.5). We demonstrate that neither priority inheritance nor priority boosting can be used as a foundation for asymptotically optimal locking protocols (Sec. 3.1) and present a novel priority boosting variant, "priority donation," that causes only $O(m)$ pi-blocking (Sec. 3.2). Using priority donation, we design the first mutex protocol for clustered JLSP scheduling (Sec. 4.1). We then show that priority donation is a general mechanism that can also be used to design suspension-based phase-fair RW (Sec. 4.2) and k-exclusion locks (Sec. 4.3). All three protocols are asymptotically optimal with regard to maximum pi-blocking under suspension-oblivious schedulability analysis.

Besides being asymptotically optimal, the bounds' constant factors, summarized in Table 1, are also small enough for the protocols to be practical. Experiments presented in Sec. 4.4 that compare our mutex protocol to the MPCP show that s-oblivious analysis is a viable alternative to s-aware analysis.

The presented protocols are the first of their kind for clustered scheduling; since clustered scheduling is a generalization of both global and partitioned scheduling, our RW and k-exclusion protocols are the first of their kind in these categories as well.

Scheduling	Constraint	Bound	Analysis
Global	Mutex	$N \cdot (2m-1)$	[8]
Partitioned	Mutex	$m + N \cdot (m-1)$	[8]
Clustered	Mutex	$m + N \cdot (m-1)$	Sec. 4.1
Clustered	RW—readers	$2m + 2N$	Sec. 4.2
Clustered	RW—writers	$2m + N(2m-1)$	Sec. 4.2
Clustered	k-exclusion	$m + N \left\lceil \frac{m-k}{k} \right\rceil$	Sec. 4.3

Table 1: OMLP per-job pi-blocking bounds in terms of the maximum number of blocking requests, where m denotes the number of processors and N denotes the number of requests issued by the job. The lower bound *per request* is $m-1$ [8]. The mutex and k-exclusion protocols are optimal within approximately a factor of two of the lower bound; the RW protocol is optimal within approximately a factor of four for writers (the lower bound does not apply to readers—see Sec. 4.5).

2. BACKGROUND AND DEFINITIONS

We consider the problem of scheduling a set of n implicit-deadline[1] sporadic tasks $\tau = \{T_1, \ldots, T_n\}$ on m processors $P_1, \ldots, P_m$. 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. T_i's *utilization* u_i is given by the fraction $u_i = e_i/p_i$. $J_{i,j}$ 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. If $j > 1$, then $a_{i,j} \geq a_{i,j-1} + p_i$. T_i is *schedulable* if it can be shown that each $J_{i,j}$ completes within p_i time units of its release. We omit the job index j if it is irrelevant and let J_i denote an arbitrary job.

A pending job can be 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 ready unless suspended by a locking protocol.

Scheduling. Under *clustered scheduling* [2, 11], 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 $c = 1$ (resp.). Each task is statically assigned to a cluster. Jobs may migrate freely within clusters, but not across cluster boundaries.

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

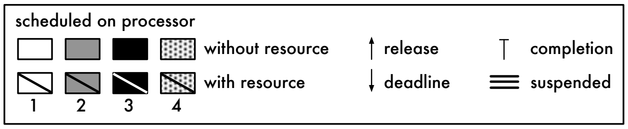


Figure 1: The notation used in subsequent example schedules.

We assume that, within each cluster, jobs are scheduled from a single ready queue using a work-conserving JLSP policy [12]. A JLSP policy assigns each job a fixed *base priority*. However, a job's *effective priority* may temporarily exceed its base priority when raised by a locking protocol (see below). Within 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-index tasks, *i.e.*, priorities are unique. We consider *global, partitioned, and clustered EDF* (GEDF, PEDF, and CEDF, resp.) as representative algorithms of this class.

Resources. The system contains r *shared resources* $\ell_1, \ldots, \ell_r$ (such as shared data objects and I/O devices) besides the m processors. When a job J_i requires a resource ℓ_q, it *issues* a *request* $\mathcal{R}$ for ℓ_q. $\mathcal{R}$ 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[3] before it releases ℓ_q. We 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, where $L_{i,q} = 0$ if $N_{i,q} = 0$.

We assume that jobs request or hold at most one resource at any time (nesting can be supported with group locks as in the FMLP [5], albeit at the expense of reduced parallelism) and that tasks do not hold resources across job boundaries.

Each resource is subject to a sharing constraint. *Mutual exclusion* of requests is required for *serially-reusable* resources, which may be held by at most one job at any time. *Reader-writer exclusion* is sufficient if a resources's state can be observed without affecting it: only *write requests* (*i.e.*, state changes) are exclusive and multiple *read requests* may be satisfied simultaneously. Resources of which there are k identical replicas (*e.g.*, graphics processing units (GPUs)) are subject to a k-*exclusion* constraint: each replica is only serially reusable and thus requires mutual exclusion,[4] but up to k requests may be satisfied at the same time by delegating them to different replicas. We let k_q denote the number of replicas of resource ℓ_q.

Locking protocols. In each case, a *locking protocol* must be employed to order conflicting requests. If a request $\mathcal{R}$ 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 $\mathcal{R}$ to be satisfied. In this paper, we focus on protocols in which waiting jobs relinquish their processor and suspend. The *request span* of $\mathcal{R}$ starts when $\mathcal{R}$ is issued and lasts until it completes, *i.e.*, it includes the request length and any acquisition delay.

Locking protocols may temporarily raise a job's effective priority. Under *priority inheritance* [19, 21], 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 waiting for ℓ_q. Alternatively, under *priority boosting* [7, 8, 17, 18, 19, 20], a resource-holding job's priority is unconditionally elevated above the highest-possible base (*i.e.*, non-boosted) priority to expedite the request completion.

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

[4]One could also consider replicated resources with RW constraints, but we are not aware of applications where such constraints arise.

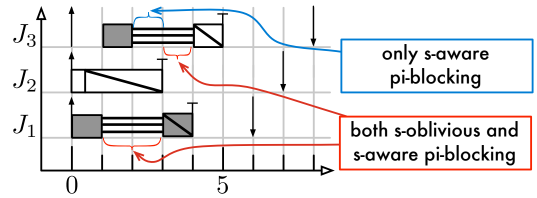


Figure 2: Example of s-oblivious and s-aware pi-blocking of three jobs sharing one resource on two GEDF-scheduled processors. J_1 suffers acquisition delay during $[1, 3)$, and since no higher-priority jobs exist it is pi-blocked under either definition. J_3, suspended during $[2, 4)$, suffers pi-blocking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

Pi-blocking. When locks are used, bounds on *priority inversion blocking* (*pi-blocking*) are required during schedulability analysis. Pi-blocking occurs when a job is delayed and this delay cannot be attributed to higher-priority demand (formalized below). We let b_i denote a bound on the total pi-blocking incurred by any J_i.

As noted in [8], there are two notions of "priority inversion" on a multiprocessor. The reason is that multiprocessor schedulability analysis has not yet matured to the point that suspensions can be analyzed under all schedulers. In particular, none of the major GEDF hard real-time schedulability tests inherently accounts for suspensions (see [3] for a recent overview). Such analysis is *suspension-oblivious* (*s-oblivious*): jobs may suspend, but each e_i must be inflated by b_i prior to applying the test to account for all additional delays. This approach is safe—converting execution time to idle time does not increase response times—but pessimistic, as even suspended jobs are (implicitly) considered to prevent lower-priority jobs from being scheduled. In contrast, *suspension-aware* (*s-aware*) schedulability analysis that explicitly accounts for b_i is available for select schedulers (*e.g.*, PSP [17, 19]). Notably, suspended jobs are *not* considered to occupy a processor under s-aware analysis.

Consequently, priority inversion is defined differently under s-aware and s-oblivious analysis: since suspended jobs are counted as demand under s-oblivious analysis, the mere *presence* of m higher-priority jobs rules out a priority inversion, whereas at least m *ready* higher-priority jobs are needed to nullify a priority inversion under s-aware analysis.

Definition 1. Under **s-oblivious** (**s-aware**) schedulability analysis, a job J_i incurs *s-oblivious* (*s-aware*) *pi-blocking* at time t if J_i is pending but not scheduled and fewer than c higher-priority jobs are **pending** (**ready**) in T_i's assigned cluster.

In both cases, "higher-priority" is interpreted with respect to base priorities. The difference between s-oblivious and s-aware pi-blocking is illustrated in Fig. 2 (see Fig. 1 for a summary of our notation). In this paper, we focus on s-oblivious pi-blocking since we are most interested in CEDF [4, 11], for which no s-aware analysis has been developed to date.

Blocking complexity. In [8], we introduced *maximum pi-blocking*, $\max_{T_i \in \tau}\{b_i\}$, as a measure of a protocol's blocking behavior. Maximum pi-blocking reflects the per-task bounds that are required for schedulability analysis. 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 total number of requests per job. Thus, when deriving asymptotic bounds, we consider, for each

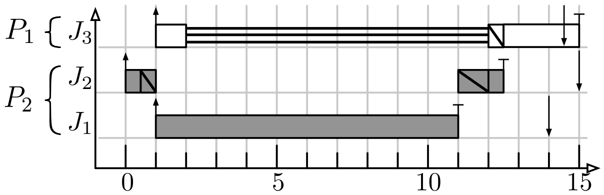


Figure 3: Example schedule of three tasks on two processors under PEDF scheduling ($c = 1$). The example shows that priority inheritance is ineffective across cluster boundaries.

T_i, $\sum_{1 \le q \le r} N_{i,q}$ and each $L_{i,q}$ to be constants and assume $n \ge m$. All other parameters are considered variable.

3. RESOURCE-HOLDER PROGRESS

The main purpose of a real-time locking protocol is to prevent maximum pi-blocking from becoming unbounded or very large (*i.e.*, bounds should not include job execution costs in addition to request lengths). This requires that resource-holding jobs progress in their execution when high-priority jobs are waiting, *i.e.*, low-priority jobs must be scheduled in spite of their low base priority when they cause other 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 job is pi-blocked) or unconditionally. As mentioned in Sec. 1, all prior protocols employ priority inheritance and priority boosting to this end—unfortunately, neither generalizes to clustered scheduling.

3.1 Limits of Inheritance and Boosting

Priority inheritance was originally developed for uniprocessor locking protocols [19, 21], but also generalizes to global scheduling [5, 8, 14]. It is a powerful aid for worst-case analysis under global scheduling because it yields the following property: if a job J_i incurs pi-blocking, and J_h holds the resource that J_i requested, then J_h is scheduled [19, 21]. Progress is thus guaranteed.

Unfortunately, priority inheritance is ineffective across cluster boundaries. For example, suppose that requests are satisfied in FIFO[5] order and priority inheritance is employed (this is essentially the global FMLP [5]). Fig. 3 depicts a schedule that may arise when this protocol is applied across clusters (where $c = 1$). J_3 misses its deadline because it incurs pi-blocking (both s-oblivious and s-aware) for virtually the *entire* duration of J_1's execution *despite* priority inheritance since J_1's deadline precedes J_3's deadline. Thus, even with priority inheritance, total pi-blocking cannot be bounded solely in terms of request lengths.

Consequently, protocols for partitioned scheduling rely on priority boosting instead of [5, 7, 8] or in addition to [17, 18, 19, 20] priority inheritance. The root cause for excessive pi-blocking is later-arriving higher-priority jobs (like J_1 above) that preempt resource-holding jobs (like J_2). Priority boosting avoids this by unconditionally raising the effective priority of resource-holding jobs above that of non-resource-holding jobs: as newly-released jobs do not yet hold resources, they cannot preempt 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 [8], but this approach does not extend to $c > 1$. For example, suppose that requests are satisfied in FIFO order, and that a resource holder's priority is boosted (as under the partitioned FMLP [5]). A possible result is shown in Fig. 4: jobs in cluster C_2 repeatedly request ℓ_1 and ℓ_2 in a pattern that causes low-priority jobs ($J_2, \ldots, J_5$) in C_1 to be priority-boosted simultaneously, which causes J_1 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.

[5]FIFO ordering is actually not required for the counterexamples in this section, but is assumed for simplicity.

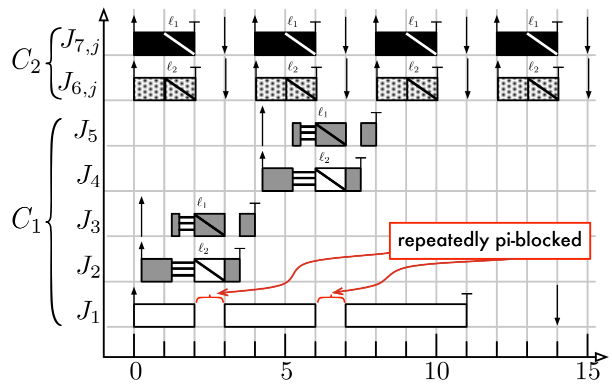


Figure 4: Example schedule of seven tasks sharing two resources (ℓ_1, ℓ_2) across two two-processor clusters under CEDF scheduling. The example shows that priority boosting may cause jobs to incur pi-blocking repeatedly if $c > 1$. If $c = 1$, then lower-priority jobs cannot issue requests while higher-priority jobs execute [8].

3.2 Priority Donation

The partitioned OMLP [8], which uses priority boosting, relies on the following two progress properties (for $c = 1$):

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 (w.r.t. 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 Fig. 4 shows, if $c > 1$, this is problematic since an "unlucky" job (like J_1) can repeatedly be a preemption "victim," 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.

Request rule. In the following, let J_i denote a job that requires a resource ℓ_q at time t_1, as illustrated in Fig. 5. 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. Priority donation achieves P1 and P2 for $1 \le c \le m$ in two steps: it first requires that J_i has a high base priority, and then ensures that J_i's effective priority remains high until J_i releases ℓ_q.

D1 J_i may issue a request only if it is among the c highest-priority pending jobs in its cluster (w.r.t. 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, *i.e.*, Property P1 holds at time

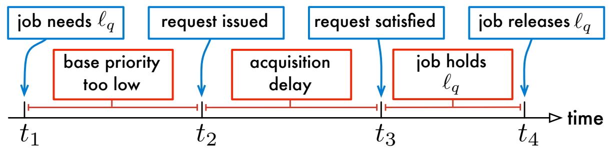


Figure 5: Illustration of the request phases under priority donation. A job J_i requires a resource ℓ_q at time t_1. J_i suspends until time t_2, when it becomes one of the c highest-priority pending jobs in its assigned cluster (Rule D1). J_i remains suspended while it suffers acquisition delay from t_2 until its request is satisfied at t_3. Priority donation ensures that J_i is continuously scheduled in $[t_3, t_4)$.

t_2 in Fig. 5. However, some—but not all—later job releases during $[t_2, t_4]$ could preempt J_1.

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, *i.e.*, 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 define how jobs become donors and when they suspend next 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 (w.r.t. 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.*, $t_a \in [t_2, t_4]$ w.r.t. J_i's request.
- **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-priority *pending* jobs (w.r.t. 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.
- **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, *i.e.*, J_d suspends and remains pending until it is no longer a priority donor.

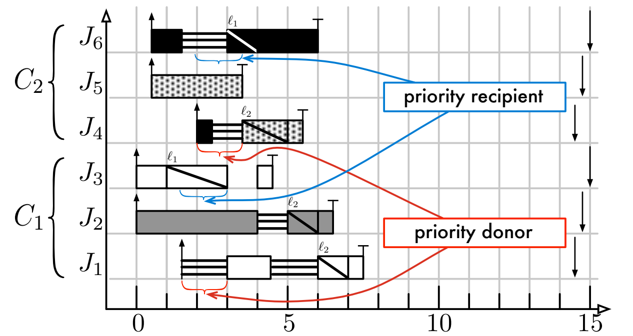


Figure 6: Schedule of six tasks sharing two serially-reusable resources across two two-processor clusters under CEDF scheduling. Under the clustered OMLP, progress is ensured with priority donation (Sec. 3) and jobs wait in FIFO order (Sec. 4.1).

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), **(b)** J_i's base priority becomes one of the c highest (w.r.t. 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.

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

Example. Fig. 6 shows a resulting schedule assuming jobs wait in FIFO order. Priority donation occurs first at time 1.5, when the release of J_1 displaces J_3 from the set of the c highest-priority jobs in C_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 3 (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 1.5, 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 2 (Rule D2). Since J_6 is already suspended, Rule D5 does not apply and J_4 remains ready. However, at time 2.5, 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 3.5 since J_5 finishes, which causes J_6 to become one of the two highest-priority pending jobs in C_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 3, thereby violating P1.

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

LEMMA 1. *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 (w.r.t. 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 2. *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), *i.e.*, 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 the duration of at most one entire request span.

Case (*ii*). Let J_x denote a job that is ready and among the c highest-priority pending jobs (w.r.t. base priorities) in cluster C_j, but not scheduled. Let A denote the set of ready jobs in C_j with higher base priorities than J_x, and let B denote the set of ready jobs in C_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 $|A|+|D| \geq c$. By the definition of B, every job in D has a base priority that exceeds J_x's base priority. Rules D5 and D6 imply that no job in D is ready (since every job in B is ready): $A \cap D = \emptyset$. Every job in D is pending (Rule D8), and every job in A is ready and hence also pending. Thus, there exist at least c pending jobs with higher base priority than J_x in C_j. Contradiction. □

Priority donation further limits maximum concurrency, which is key to the analysis in the remainder of this paper.

LEMMA 3. *Let $R_j(t)$ denote the number of requests issued by jobs in cluster C_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 jobs in C_j (at time t w.r.t. base priorities), and let I denote the set of jobs in C_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 1–3 provide a strong foundation that enables the design of simple, yet asymptotically optimal, locking protocols.

4. THE CLUSTERED OMLP

The *O(m) locking protocol* (OMLP) [8] is a family of asymptotically optimal suspension-based multiprocessor locking protocols for JLSP schedulers, *i.e.*, member protocols cause jobs to incur only $O(m)$ pi-blocking under s-oblivious analysis. In [8], we proposed two OMLP mutex protocols for global and partitioned scheduling. In this section, we augment the OMLP family with priority-donation-based mutex, reader-writer, and k-exclusion locks for clustered scheduling, and discuss how and when to combine OMLP variants.

4.1 Mutex Locks

Priority donation is a powerful aid for worst-case analysis. This is witnessed by the simplicity of the following mutex protocol for clustered scheduling, which uses simple FIFO queues. In contrast, the global and partitioned OMLP mutex protocols, which rely on priority inheritance and priority boosting (resp.), each require a combination of priority and FIFO queues to achieve an $O(m)$ bound.

Structure. There is a FIFO queue FQ_q for each serially-reusable resource ℓ_q. The job at the head of FQ_q holds ℓ_q.

Rules. Jobs that issue conflicting requests are serialized with FQ_q. Let J_i denote a job that issues a request $\mathcal{R}$ for ℓ_q.

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

Rules X1–X3 correspond to times t_2–t_4 in Fig. 5.

Example. Fig. 6 depicts an example of the clustered OMLP for serially-reusable resources. J_3 requests ℓ_1 at time 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 1.5, it is appended to FQ_1 and suspends. When J_3 releases ℓ_1 at time 3, J_6 becomes the new head of FQ_1 and resumes (Rule X3).

At time 3.5, 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 4 and 4.5. Importantly, priorities are ignored in each FQ_q: when J_4 releases ℓ_2 at time 5, 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 may yield $\Omega(mn)$ pi-blocking [8].[6]

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

LEMMA 4. *At most m jobs are enqueued in any FQ_q.*

PROOF. By Lemma 3, 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 some FQ_q. □

LEMMA 5. *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 4, at most $m-1$ other jobs precede J_i in FQ_q. By Lemma 1, the job at the head of FQ_q is always scheduled. Therefore, J_i becomes the head of FQ_q after the combined length of $m-1$ 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 $b_i = m \cdot L^{max} + \sum_{q=1}^{r} N_{i,q} \cdot (m-1) \cdot L^{max} = O(m)$ s-oblivious pi-blocking.*

[6][8] shows $\Omega(mn)$ *s-aware* pi-blocking, but it is trivial to obtain an s-oblivious bound by isolating a task in a dedicated cluster.

PROOF. By Lemma 2, the duration of s-oblivious pi-blocking caused by priority donation is bounded by the maximum request span. By Lemma 5, 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 Sec. 2 that $\sum_{q=1}^{r} N_{i,q}$ and L^{max} are constant. The bound follows. □

The protocol for serially-reusable resources is thus asymptotically optimal w.r.t. maximum s-oblivious pi-blocking.

4.2 Reader-Writer Locks

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, *i.e.*, 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 [9]. 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 [9], we introduced *phase-fair* RW locks as an alternative, under which *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, whereas one write request is satisfied at the beginning of a writer phase. This results in $O(1)$ acquisition delay for read requests without starving write requests. We presented spin-based phase-fair RW locks in [9]. With priority donation as a base, we can transfer the concept to suspension-based locks.

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, *i.e.*, the designations "collecting" and "draining" are not static.

Reader rules. Let J_i denote a job that issues a read request $\mathcal{R}$ 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 $\mathcal{R}$. If WQ_q is non-empty, then J_i suspends.
- **R2** $\mathcal{R}$ is satisfied either immediately if WQ_q is empty when $\mathcal{R}$ is issued, or when J_i is subsequently resumed.
- **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 $\mathcal{R}$ 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).

Writer rules. Let J_w denote a job that issues a write request $\mathcal{R}$ 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.

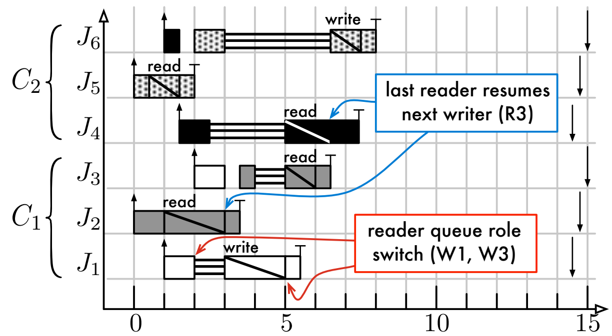


Figure 7: Schedule of six tasks sharing one RW resource across two two-processor clusters under CEDF scheduling.

- **W1** J_w is enqueued in WQ_q when it issues $\mathcal{R}$. J_w suspends until $\mathcal{R}$ is satisfied unless WQ_q and CQ_q are both empty at the time of request. If WQ_q is empty and CQ_q is not, then the roles of CQ_q and DQ_q are switched.
- **W2** $\mathcal{R}$ is satisfied either immediately if WQ_q and CQ_q are both empty when $\mathcal{R}$ is issued, or when J_w is resumed.
- **W3** J_w is dequeued from WQ_q when $\mathcal{R}$ 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, the roles of CQ_q and DQ_q are switched.

Rules R1–R3 and W1–W3 correspond to times t_2–t_4 in Fig. 5 (resp.), and are illustrated in Fig. 7.

Example. The resource ℓ_1 is first read by J_5, which is enqueued in RQ_q^1, the initial collecting queue, at time 0.5 (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 2. Since CQ_1 is non-empty, the roles of CQ_1 and DQ_1 are switched, *i.e.*, RQ_q^1 becomes the draining reader queue, and J_1 suspends (Rule W1). J_4 issues a read request soon thereafter and is enqueued in RQ_q^2 (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 3, 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_q^2 (J_3 and J_4). Since WQ_1 is non-empty (J_6 was enqueued at time 3), RQ_q^2 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 6.5 and *all* waiting readers are resumed at the beginning of the next reader phase at time 5 (Rule W3).

Analysis. 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 6. *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.*

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, 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 3), and each job holding ℓ_q is scheduled (Lemma 1). The claim follows. □

LEMMA 7. *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 write 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 6, 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 1). Therefore, J_i waits for the duration of at most one read and one write request. □

Lemma 7 shows that readers incur $O(1)$ acquisition delay. Next, we show that writers incur $O(m)$ acquisition delay.

LEMMA 8. *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 3 that at most $m-1$ other jobs precede J_w in WQ_q (analogously to Lemma 4). By Lemma 1, J_w's predecessors together hold ℓ_q for the duration of at most $m-1$ write requests. By Lemma 6, 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 w.r.t. maximum s-oblivious pi-blocking.

THEOREM 2. *The clustered OMLP for RW resources causes a job J_i to incur at most $b_i = 2 \cdot m \cdot L^{max} + \sum_{q=1}^{r} N_{i,q} \cdot (2 \cdot m - 1) \cdot L^{max} = O(m)$ s-oblivious pi-blocking.*

PROOF. By Lemma 2, the duration of s-oblivious pi-blocking caused by priority donation is bounded by the maximum request span. By Lemma 8, maximum acquisition delay per write request is bounded by $(2m-1) \cdot L^{max}$; by Lemma 7, maximum acquisition delay per read request is bounded by $2 \cdot L^{max}$. The maximum request span is thus bounded by $2m \cdot L^{max}$. Recall from Sec. 2 that $\sum_{q=1}^{r} N_{i,q}$ and L^{max} are constant. The bound follows. □

Since priority inheritance is sufficient for the global OMLP mutex protocol from [8], one might wonder if it is possible to apply the same design using priority inheritance instead of priority donation to obtain an $O(m)$ RW protocol under global scheduling. 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 with priority inheritance may cause $\Omega(n/c)$ s-oblivious pi-blocking since it is possible to construct schedules that are conceptually similar to the one shown in Fig. 4. This demonstrates the power of priority donation, and also highlights the value of the clustered OMLP even for the special cases $c = m$ and $c = 1$.

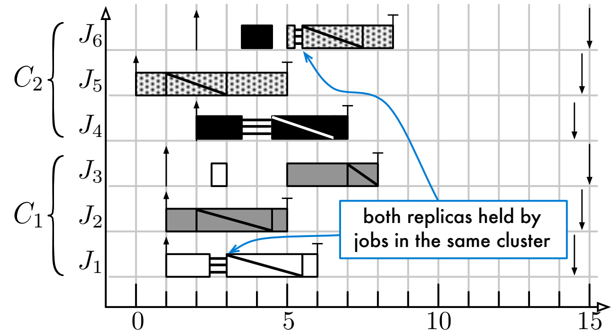


Figure 8: Six tasks sharing two instances of one resource across two two-processor clusters under CEDF scheduling.

4.3 k-Exclusion Locks

For some resource types, one option to reduce contention is to *replicate* them. For example, if potential overload of a co-processor for digital signal processing (DSP) 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 instance, or every task may request any instance. The former approach, which corresponds to partitioned scheduling, has the advantage that a mutex protocol suffices, but it also implies that some instances may idle while jobs wait to acquire their designated instance. 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 is 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 Sec. 4.5 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 $RS_q \neq \emptyset$, then KQ_q is empty.

Rules. Let J_i denote a job that issues a request $\mathcal{R}$ for ℓ_q.

- **K1** If $RS_q \neq \emptyset$, then J_i acquires an idle replica from RS_q. Otherwise, J_i is enqueued in KQ_q and suspends.
- **K2** $\mathcal{R}$ is satisfied either immediately (if $RS_q \neq \emptyset$ at the time of request) or when J_i is removed from KQ_q.
- **K3** If KQ_q is non-empty when $\mathcal{R}$ 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 Fig. 5.

Example. Fig. 8 depicts an example schedule for one resource (ℓ_1) with $k_1 = 2$. J_5 obtains a replica from RS_1 at time 1 (Rule K1). The second instance of ℓ_1 is acquired by J_2 at time 2. As RS_1 is now empty, J_1 is enqueued in KQ_1 and suspends when it requests ℓ_1 at time 2.5. However, it is soon resumed when J_5 releases its replica at time 3 (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 to a mutex constraint), then J_1 would have continued to wait while C_2's instance would have idled. This happens again at time 5.5: since no job in C_1 requires ℓ_1 at the time, both instances are used by jobs in C_2.

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

LEMMA 9. *At most $m - k_q$ jobs are enqueued in KQ_q.*

PROOF. Lemma 3 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 10. *Let J_i denote a job that issues a request $\mathcal{R}$ 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 9, 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 1), requests are satisfied at a rate of at least k_q requests per maximum request length until $\mathcal{R}$ is satisfied. The stated bound follows. □

Lemma 10 shows that J_i incurs at most $O(\frac{m}{k_q})$ pi-blocking per request (and none if $k_q = m$), which implies asymptotic optimality w.r.t. maximum s-oblivious pi-blocking.

THEOREM 3. *The clustered OMLP for replicated resources causes a job J_i to incur at most $b_i = m \cdot L^{max} + \sum_{q=1}^{r} N_{i,q} \cdot \lceil (m - k_q)/k_q \rceil \cdot L^{max} = O(m)$ s-oblivious pi-blocking.*

PROOF. By Lemma 10, maximum acquisition delay per request for ℓ_q is bounded by $\lceil (m - k_q)/k_q \rceil \cdot L^{max}$. Since $\min_{1 \leq q \leq r} k_q \geq 1$, the maximum request span is thus bounded by $(\lceil (m-1)/1 \rceil + 1) \cdot L^{max} = m \cdot L^{max}$. Lemma 2 limits the duration of s-oblivious pi-blocking due to priority donation to the maximum request span. The bound follows since $\sum_{q=1}^{r} N_{i,q}$ and L^{max} are constant. □

4.4 Experiments

Asymptotic optimality does not necessarily translate into better schedulability in practice. To provide some sense of the practical viability of our locking protocols under s-oblivious analysis, we present results from preliminary experiments that were conducted to compare one of these protocols to s-aware alternatives.[7] 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 [17, 18, 19] and PSP scheduling is considered to be the de facto standard. For the MPCP, accurate s-aware schedulability analysis exists [17]. We experimentally compared schedulability under our proposed mutex protocol when used under PEDF scheduling to two variants of this standard approach, namely the original suspension-based variant [18, 19] and a newer variant based on "virtual spinning" [17], where "spinning" jobs do in fact suspend but other local jobs may not issue requests until the "spinning" job's request is satisfied. The goal of these experiments is *not* to claim that s-oblivious analysis is superior. Rather, they show that it is a practical alternative.

Setup. Each task set was generated based on three parameters: a *per-processor utilization* U, a *resource probability* $\mathcal{P}_{res}$, and a *request probability* $\mathcal{P}_{req}$. On each processor, we generated tasks until reaching U by choosing $u_i \in [0, 1]$ using an exponential distribution with mean 0.1 and by uniformly choosing $p_i \in [10\text{ms}, 100\text{ms}]$. The number of resources r was generated randomly using a geometric distribution such that $r = x$ with probability $(1 - \mathcal{P}_{res}) \cdot \mathcal{P}_{res}^{x-1}$, where $r \geq 1$. Similarly, each $N_{i,q}$, where $N_{i,q} \geq 0$, was randomly chosen from a geometric distribution based on $\mathcal{P}_{req}$. Following Lakshmanan *et al.* [17], we chose $L_{i,q} \in [5\mu s, 1280\mu s]$. As requests are frequently short in practice [6, 10], we chose each $L_{i,q}$ using an exponential distribution with mean $20\mu s$.

For each $m \in \{4, 8, 16, 24, 32\}$, we varied $U \in [0.25, 0.95]$ in steps of 0.025, $\mathcal{P}_{res} \in [0.05, 0.95]$ in steps of 0.05, and $\mathcal{P}_{req} \in [0.05, 0.75]$ in steps of 0.05. We tested 100 task sets for each combination of m, U, $\mathcal{P}_{req}$, and $\mathcal{P}_{req}$, for a total of over 4.2 million task sets. When visualized as a function of U, this results in 1,425 graphs, of which two representative examples[8] are shown in Fig. 9.

Results. Fig. 9(a) shows schedulability, *i.e.*, the fraction of task sets that can be shown to never miss a deadline, as a function of U on eight processors. Surprisingly, the OMLP under s-oblivious analysis yields higher schedulability than either MPCP variant—which is in fact the case for all graphs for $m = 4$ and $m = 8$. Fig. 9(b) shows an example for $m = 16$. Here, the suspension-based MPCP and the OMLP perform similarly. Overall, the OMLP's competitiveness decreases with increasing m, which reflects its $O(m)$ bound. The suspension-based MPCP outperforms the OMLP for $m = 24$ and $m = 32$. However, the OMLP never performs worse than the virtual-spin-based MPCP variant in any of the tested configurations.

Our results show that **(i)** s-oblivious analysis is a viable approach until superior s-aware analysis is developed and **(ii)** s-oblivious analysis provides a competitive baseline against which future s-aware analysis and protocols should be compared. Clearly, it would be desirable to develop effective s-aware schedulability analysis for all practical scheduling algorithms. However, it should be noted that accurate analysis of task suspensions is a notoriously difficult problem. Barring unforeseeable breakthroughs, *efficient* s-aware analysis may not be forthcoming for all global, and hence clustered, JLSP scheduling algorithms in the near future (or ever).

4.5 Protocol Combinations, Limitations, and Open Questions

The clustered mutex protocol (Sec. 4.1) generalizes the partitioned OMLP from [8] in terms of blocking behavior; there is thus little reason to use both in the same system.

The global OMLP from [8] cannot be used with the protocols in this paper since priority inheritance is incompatible with priority donation (from an analytical point of view). Both mutex protocols have an $O(m)$ s-oblivious pi-blocking bound, but differ in constant factors and w.r.t. which jobs incur pi-blocking. Specifically, 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 plan to explore this tradeoff in future work.

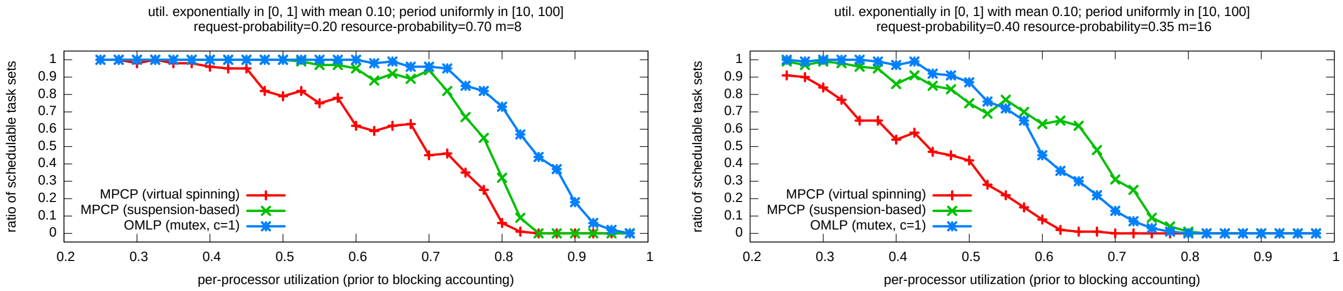
The protocols presented in this paper can be freely combined since they all rely on priority donation and because their protocol rules do not conflict. However, care must be taken to correctly identify the maximum request span.

Optimality of relaxed-exclusion protocols. Under phase-fair RW locks (Sec. 4.2), read requests incur at most $O(1)$ acquisition delay. Similarly, requests incur only $O(m/k_q)$ acquisition delay under the k-exclusion protocol (Sec. 4.2). Yet, we only prove $O(m)$ *maximum* s-oblivious pi-blocking bounds—since both relaxed-exclusion constraints generalize mutual exclusion, this is unavoidable [8].

However, as noted above, *any* job may become a priority donor and thus suspend (at most once) for the duration of the maximum

[7]It is not possible to give exhaustive experimental results due to space constraints. An implementation-based comparison of locking protocols that considers overheads is currently under preparation.

[8]An online appendix with all graphs is available at: http://www.cs.unc.edu/ anderson/papers.html.



(a) $m = 8$, $\mathcal{P}_{req} = 0.2$, and $\mathcal{P}_{res} = 0.7$

(b) $m = 16$, $\mathcal{P}_{req} = 0.4$, and $\mathcal{P}_{res} = 0.35$

Figure 9: Schedulability as a function of per-processor utilization U. S-oblivious analysis is competitive, and especially so for $m \leq 8$. Per-resource contention increases with increasing values of $\mathcal{P}_{req}$ and U; the number of requests per job increases with $\mathcal{P}_{res}$ and $\mathcal{P}_{req}$.

request span. 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 unexplored topic that warrants further attention.

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. 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 cause processor demand under s-oblivious analysis). S-aware analysis is essential to derive analytical benefits from highly replicated resources.

5. CONCLUSION

Existing global and partitioned suspension-based real-time locking protocols do not generalize to clustered scheduling due to the unique combination of both global and partitioned aspects. To overcome this, we designed priority donation, a novel mechanism for ensuring resource-holder progress that works for $1 \leq c \leq m$.

Using priority donation as a foundation, we augmented the OMLP family of locking protocols with three suspension-based real-time locking protocols for clustered JLSP schedulers that realize mutex, RW, and k-exclusion constraints. The two latter 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). Each protocol is asymptotically optimal w.r.t. maximum s-oblivious pi-blocking. The 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 partitioned and global scheduling as well.

In future algorithmic work, we would like to extend our results to s-aware schedulability analysis and explore the optimality of relaxed-exclusion protocols. In future empirical work, we plan to evaluate the OMLP family in LITMUS$^{\mathrm{RT}}$.

Acknowledgement. We thank Glenn Elliott for helpful discussions concerning k-exclusion constraints and GPUs.

6. REFERENCES

[1] T. Baker. Stack-based scheduling for realtime processes. *Real-Time Sys.*, 3(1):67–99, 1991.
[2] T. Baker and S. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In *Handbook of Real-Time and Embedded Sys.*, Chapman Hall/CRC, Boca Raton, Florida, 2007.
[3] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, and S. Stiller. Improved multiprocessor global schedulability analysis. *Real-Time Sys.*, 46(1):3–24, 2010.
[4] A. Bastoni, B. Brandenburg, and J. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor real-time schedulers. In *Proc. RTSS*, 2010.
[5] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proc. RTCSA*, 2007.
[6] B. Brandenburg and J. Anderson. A comparison of the M-PCP, D-PCP, and FMLP on LITMUS$^{\mathrm{RT}}$. In *Proc. OPODIS*, 2008.
[7] B. Brandenburg and J. Anderson. An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS$^{\mathrm{RT}}$. In *Proc. RTCSA*, 2008.
[8] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proc. RTSS*, 2010.
[9] B. Brandenburg and J. Anderson. Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-Time Sys.*, 46(1):25–87, 2010.
[10] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Synchronization on real-time multiprocessors: To block or not to block, to suspend or spin? In *Proc. RTAS*, 2008.
[11] J. Calandrino, J. Anderson, and D. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proc. ECRTS*, 2007.
[12] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Chapman and Hall/CRC, 2004.
[13] C. Chen and S. Tripathi. Multiprocessor priority ceiling based protocols. Technical Report CS-TR-3252, Univ. of Maryland, 1994.
[14] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *Proc. RTSS*, 2009.
[15] Dario Faggioli, Giuseppe Lipari, and Tommaso Cucinotta. The multiprocessor bandwidth inheritance protocol. In *Proc. ECRTS*, 2010.
[16] P. Gai, M. di Natale, G. Lipari, A. Ferrari, C.Gabellini, and P. Marceca. A comparison of MPCP and MSRP when sharing resources in the Janus multiple processor on a chip platform. In *Proc. RTAS*, 2003.
[17] K. Lakshmanan, D. Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Proc. RTSS*, 2009.
[18] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proc. ICDCS*, 1990.
[19] R. Rajkumar. *Synchronization In Real-Time Sys.—A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.
[20] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proc. RTSS*, 1988.
[21] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Trans. on Computers*, 39(9):1175–1185, 1990.