A Note on Blocking Optimality in Distributed Real-Time Locking Protocols

Björn B. Brandenburg
MPI-SWS

Initial version: July 2012 / Revised: April 2013

Abstract

Lower and upper bounds on *maximum priority inversion blocking* (*pi-blocking*) are established under distributed multiprocessor real-time semaphore protocols (where resources may be accessed only from specific synchronization processors). Prior work on shared-memory multiprocessor semaphore protocols (which require resources to be accessible from potentially any processor) has established bounds of $\Theta(m)$ and $\Theta(n)$ maximum pi-blocking under suspension-oblivious and suspension-aware analysis, respectively, where m denotes the total number of processors and n denotes the number of tasks. In this paper, it is shown that in the case of distributed semaphore protocols, there exist two different task allocation scenarios that give rise to distinct lower bounds. In the case of *co-hosted* task allocation, where application tasks may also be assigned to synchronization processors, $\Omega(\Phi \cdot n)$ maximum pi-blocking is unavoidable for some tasks under any locking protocol under *both* suspension-aware and suspension-oblivious schedulability analysis. In contrast, in the case of *disjoint synchronization and application processors* (*i.e.*, if application tasks are not scheduled on synchronization processors), only $\Omega(m)$ and $\Omega(n)$ maximum pi-blocking is fundamental under suspension-oblivious and suspension-aware analysis, respectively, as in the shared-memory case. These bounds are shown to be asymptotically tight with the construction of two new distributed real-time semaphore protocols that ensure $O(m)$ and $O(n)$ maximum pi-blocking under suspension-oblivious and suspension-aware analysis, respectively.

1 Introduction

The goal of a real-time locking protocol is to provide *predictable* access to shared resources subject to mutual exclusion constraints, that is, a real-time

locking protocol must allow worst-case blocking to be bounded *a priori.* As excessive locking-related delays can endanger a real-time time task's temporal correctness, the primary objective of a good real-time locking protocol is to avoid such blocking to the extent possible. This immediately leads to questions of optimality: if some blocking is unavoidable when using locks, then *what is the minimal bound on worst-case blocking that any locking protocol can guarantee?* In other words, which upper bounds on worst-case blocking must a real-time locking protocol ensure to be deemed (asymptotically) optimal? This question has long been answered for *uniprocessor* locking protocols [2, 20, 23], and has recently been answered for the case of *shared-memory* multiprocessor semaphore protocols ([11–13, 24]), where shared resources may be accessed from potentially any processor. However, for the case of *distributed* multiprocessor semaphore protocols, where each resource may only be accessed from a specific (subset of) processor(s), this fundamental question has not been studied in prior work.

Contributions. In this paper, we address this gap in the understanding of multiprocessor real-time synchronization. In particular, we show that there exist in fact two distinct cases, termed *co-hosted* and *disjoint* task and resource allocation, which lead to different lower bounds on unavoidable blocking. Notably, we show that blocking in the co-hosted scenario, where real-time tasks are allocated to *synchronization processors* (*i.e.*, to processors that have access to shared resources), can be asymptotically worse than in the comparable shared-memory case by a factor of Φ, where Φ denotes the ratio of the maximum response time and the minimum period (formalized in Sec. 2 below). In contrast, in the disjoint case, where synchronization processors are prohibited from hosting real-time tasks, blocking that is asymptotically not worse than in the comparable shared-memory case is possible. We show these bounds to be asymptotically tight by constructing two distributed real-time semaphore protocols that ensure asymptotically minimal blocking under so-called *suspension-oblivious* and *suspension-aware* schedulability analysis, respectively, which are two concepts central to blocking optimality that we review in detail in the next section. Finally, we demonstrate that, asymptotically speaking, prior distributed real-time semaphore protocols do not ensure minimal blocking under either type of schedulability analysis.

2 Background and Definitions

In this section, we establish key definitions and review prior results. In short, the results presented in this paper apply to arbitrary deadline sporadic tasks provisioned on a multiprocessor platform with non-uniform processor clusters. Non-processor resources are accessible only from select processors and may be accessed using *remote procedure calls* (RPCs). We formalize these assumptions as follows.

2.1 System Model

We consider the problem of scheduling a set of n sporadic real-time tasks $\tau = \{T_1, \dots, T_n\}$ on a set of m processors. Each sporadic task T_i is characterized by its *minimum inter-arrival separation* p_i, its per-job *worst-case execution time* e_i, and its *relative deadline* d_i, where $e_i \leq \min(d_i, p_i)$.

We let J_i denote an arbitrary job of task T_i. A job is *pending* from its release until it completes, and while it is pending, it is either *ready* and may be scheduled on a processor, or *suspended* and not available for scheduling. A job J_i released at time t_a has its *absolute deadline* at time $t_a + d_i$.

A task's *maximum response time* r_i describes the maximum time that any J_i remains pending. A task T_i is *schedulable* if it can be shown that $r_i \leq d_i$; the set of all tasks τ is schedulable if each $T_i \in \tau$ is schedulable. We define Φ to be the ratio of the maximum response time and the minimum period; formally

$$\Phi = \frac{\max_i\{r_i\}}{\min_i\{p_i\}}. \tag{1}$$

Clusters. The set of m processors consists of K pairwise disjoint *clusters* (or sets) of processors, where $2 \leq K \leq m$. We let C_j denote the jth cluster, and let m_j denote the number of processors in C_j, where $\sum_{j=1}^{K} m_j = m$. A common special case is a *partitioned* system, where $m_j = 1$ for each C_j. However, in general, clusters do *not* necessarily have a uniform size. We preclude the special case of $K = 1$ and $m_1 = m$, that is, the special case of a single globally scheduled cluster, because distributed locking protocols are relevant only if there are at least two clusters.

For notational convenience, we assume that clusters are indexed in order of non-decreasing cluster size: $m_j \leq m_{j+1}$ for $1 \leq j < K$. In particular, m_1 denotes the (or one of the) smallest cluster(s) in the system (with ties broken arbitrarily). Since $K \geq 2$, we have $m_1 \leq \frac{m}{2}$. This fact is exploited by the lower-bound argument in Sec. 3.

Scheduling. Each task T_i is statically assigned to one of the K clusters; we let $C(T_i)$ denote the cluster to which T_i has been assigned. Each cluster is scheduled independently according to a *job-level fixed-priority* (JLFP) scheduling policy [14]. A JLFP scheduler assigns each pending job a fixed priority and, at any point in time, schedules the m_j highest-priority ready jobs (or agents, see below) in each cluster C_j. Jobs may freely migrate among processors belonging to the same cluster (*i.e.*, global JLFP scheduling is used within each cluster), but jobs may not migrate across cluster boundaries. Note that this does not preclude the partitioned scheduling of shared-memory systems since it is possible to interpret each processor as a singleton cluster.

Two commonly used JLFP policies are *fixed-priority* (FP) scheduling, where each task is assigned a fixed priority and each job uses the priority of its task, and *earliest-deadline first* (EDF) scheduling, where jobs are prioritized in order of decreasing absolute deadlines (with ties broken arbitrarily).

A different JLFP policy may be use in each cluster. Our results apply to any JLFP policy.

2.2 Distributed Real-Time Semaphore Protocols

Tasks may share serially reusable resources (such as co-processors, I/O ports, or shared data structures). Access to *shared resources* is governed by a *locking protocol* that ensures mutual exclusion. In this paper, we focus on *semaphore* protocols, that is, on locking protocols in which blocked tasks wait by suspending (in contrast to *spin-based* protocols, in which tasks busy-wait by executing a delay loop).

There are two major types of semaphore protocols. In *shared-memory* locking protocols, tasks execute critical sections *locally*, in the sense that each task accesses shared resources directly from the processor on which it is scheduled, such that, over time, resources are accessed from multiple processors. For example, this is the case under the classic *Multiprocessor Priority-Ceiling Protocol* (MPCP) [19, 20].

In contrast, in *distributed* locking protocols, each resource is accessed only from a designated *synchronization processor* (or *cluster*). Such protocols, which derive their name from the fact that they could also be used in distributed systems (*i.e.*, in the absence of shared memory), require critical sections to be executed *remotely* if tasks access resources not local to their assigned processor.

In this paper, we focus on distributed real-time semaphore protocols.

Resource model. The tasks in τ are assumed to share n_r resources (besides the processors). Each shared resource ℓ_q (where $1 \leq q \leq n_r$) is *local* to exactly one of the K clusters (but can be accessed from any cluster using RPC invocations). We let $C(\ell_q)$ denote the cluster to which ℓ_q is local. Cluster $C(\ell_q)$ is also called the *synchronization cluster* for ℓ_q.

To facilitate resource access by remote tasks, access to each shared resource is mediated by one or more *resource agents*. To access a shared resource ℓ_q, a job J_i invokes an agent on cluster $C(\ell_q)$ using synchronous RPC to carry out a request on J_i's behalf. An agent is *active* while it is processing requests, and *inactive* otherwise. While active, an agent is either ready or suspended (*e.g.*, if the current request involves asynchronous I/O operations). After issuing a request for a resource, J_i suspends until notified by the invoked agent that the request has been carried out. A *locking protocol* determines how concurrent, conflicting requests are serialized. We review a classic protocol for this purpose in Sec. 2.2.1 below.

We let $N_{i,q}$ denote the maximum number of times that any J_i accesses ℓ_q, and let $L_{i,q}$ denote the corresponding *maximum critical section length*, that is, the maximum time that the agent handling J_i's RPC invocation requires exclusive access to ℓ_q to carry out the invoked operation (where $L_{i,q} = 0$ if $N_{i,q} = 0$). For notational convenience, we define

$$L^{max} \triangleq \max\{L_{i,q} \mid 1 \leq q \leq n_r \;\wedge\; T_i \in \tau\}. \tag{2}$$

Jobs invoke at most one agent at any time, and agents do not invoke other agents as part of handling a resource request.

We initially assume that jobs can invoke agents without any delay, that is, the overhead of cluster-to-cluster communication is assumed negligible. If a distributed locking protocol is implemented on top of a global shared memory (*e.g.*, see [10]), this assumption is realistic. However, if RPC invocations must be routed across a shared interconnect (*e.g.*, this is the case in some emerging many-core platforms), then additional communication delays must be accounted for. However, these communication delays do not affect the blocking analysis (*i.e.*, they do not affect the contention for shared resources) and thus can be ignored when deriving asymptotic bounds. We revisit the issue of non-negligible communication delays in Sec. 6.

Finally, in a real system, there likely exist resources in each cluster that are shared only among local tasks. Such *local resources* can be handled using shared-memory protocols (or uniprocessor protocols in the case of singleton clusters, *i.e.*, if $m_j = 1$) and are not subject of this paper. For each resource ℓ_q, we assume that it is accessed by tasks from at least two different clusters.

2.2.1 The Distributed Priority Ceiling Protocol

Rajkumar *et al.* were first to study real-time locking on multiprocessors and proposed the *Distributed Priority-Ceiling Protocol* (DPCP) [20, 21], which, as the name implies, does not require shared memory. A detailed review of the DPCP and other multiprocessor real-time locking protocols can be found in [13]; herein, we briefly review the most essential aspects.

The DPCP assumes *partitioned fixed-priority* (P-FP) scheduling, that is, the DPCP fundamentally requires $m_j = 1$ for each cluster (or, rather, partition) C_j. Each resource ℓ_q is assumed *local* (*i.e.*, statically assigned) to a specific processor and may not be accessed from other processors. The DPCP provides one resource agent, denoted $A_{q,i}$, for each resource ℓ_q and each task T_i. Importantly, resource agents are subject to *priority boosting*, which means that they have priorities higher than any regular task (and thus cannot be preempted by "normal" jobs), although resource agents acting on behalf of higher-priority tasks may still preempt agents acting on behalf of lower-priority tasks. That is, an agent $A_{q,h}$ may preempt another agent $A_{r,l}$ if T_h has a higher priority than T_l. After a job has invoked an agent, it suspends until its request has been carried out.

On each processor, conflicting accesses are mediated using the well-known uniprocessor *priority-ceiling protocol* (PCP) [20, 23]. The PCP assigns each resource a *priority ceiling*, which is the priority of the highest-priority task (or agent) accessing the resource, and, at runtime, maintains a *system ceiling*, which is the maximum priority ceiling of any currently locked resource. A job (or agent) is permitted to lock a resource only if its priority exceeds the current system ceiling, waiting jobs/agents are ordered by effective scheduling priority, and priority inheritance [20, 23] is applied to prevent unbounded "priority inversion" (see Sec. 2.3 below).

2.2.2 Simplified Protocol Assumptions

The DPCP can be considered the *de facto* standard distributed multiprocessor real-time semaphore locking protocol. In this paper, we abstract from the specifics of the DPCP to consider a larger class of "DPCP-like" protocols. Specifically, we focus on distributed real-time locking protocols that ensure progress by means of two properties.

- **A1** Agents are priority-boosted: agents always have a higher priority than regular jobs.
- **A2** The distributed locking protocol is *weakly work-conserving*: a resource request $\mathcal{R}$ for a resource ℓ_q is unsatisfied at time t (*i.e.*, $\mathcal{R}$ has been

issued but is not yet being processed) only if *some* resource (not necessarily ℓ_q is currently unavailable (*i.e.*, some agent is currently processing a request for any resource).

Assumption A1 is necessary to expedite request completion since excessive delays cannot generally be avoided if jobs can preempt agents. Assumption A2 rules out pathological protocols that "artificially" delay requests. We consider this form of work conservation to be "weak" because it does not require the *requested* resource to be unavailable; a request for an available resource may also be delayed if some *other* resource is currently in use. While a stronger progress guarantee may be desirable in practice, Assumption A2 suffices to establish the lower and upper bounds presented herein. Notably, the DPCP is only weakly work-conserving (and not work-conserving with respect to each requested resource) since requests for available resources may remain temporarily unsatisfied due to ceiling blocking under the PCP.

Assumptions A1 and A2 together ensure that any delay in the processing of resource requests can be attributed only to other resource requests.

Agents. Under the DPCP, jobs do not require agents to access resources local to their assigned processor since jobs can directly participate in the PCP. In a sense, this can be seen as jobs taking over the role of their agent on their local processor. To simplify the discussion in this paper, we assume herein that resources are accessed *only* via agents (*i.e.*, jobs invoke agents even for resources that happen to be local to their assigned processors). This does not change the algorithmic properties of the DPCP.

For the sake of simplicity, we further assume that there is only a single local agent for each resource. As seen in the DPCP [20, 21], it can make sense to use more than one agent per resource; however, in the following, we abstract from such protocol specifics and let a single agent A_q represent all agents corresponding to a resource ℓ_q.

2.2.3 Co-Hosted and Disjoint Task Allocation

Processor clusters that host resource agents are called *synchronization clusters*. Conversely, processor clusters that host sporadic real-time tasks are called *application clusters*. In this paper, we establish asymptotically tight lower and upper bounds on maximum pi-blocking in two separate task and resource allocation scenarios, which we refer to as "co-hosted" and the "disjoint" settings, respectively.

In a *co-hosted setting*, the set of application clusters overlaps with the set of synchronization clusters, that is, there exists a cluster that hosts both

tasks and agents. In contrast, in a *disjoint setting*, clusters may host either agents or tasks, but not both. The significance of these two settings is that they give rise to two distinct lower bounds on worst-case blocking, as will become apparent in Sec. 3.

2.3 Priority Inversion Blocking

The sharing of resources subject to mutual exclusion constraints inevitably causes some delays because conflicting concurrent requests must be serialized. Such delays are problematic in a real-time system if they lead to an increase in worst-case response times (*i.e.*, if they affect some r_i). Conversely, delays that do not affect r_i are not considered to constitute "blocking" in real-time systems. This is captured by the concept of *priority inversion* [20, 23], which, intuitively, exists if a job that should be scheduled according to its base priority is not scheduled, either because it is suspended (while waiting to gain access to a shared resource) or because a job or agent with elevated effective priority prevents it from being scheduled. To avoid confusion with other interpretations of the term "blocking" (*e.g.*, in an OS context, "blocking" often is used synonymously with suspending), we use the term *priority inversion blocking* (*pi-blocking*) to denote any resource-sharing-related delay that affects worst-case response times in this paper. We let b_i denote a bound on the *maximum pi-blocking* incurred by any job of task T_i.

2.3.1 Suspension-Oblivious vs. Suspension-Aware Analysis

Prior work has shown that there exist in fact two kinds of priority inversion [11], depending on how suspensions are accounted for by the employed schedulability analysis. The difference arises because many published schedulability tests simply assume the absence of self-suspensions, which are notoriously difficult to analyze (*e.g.*, see [22]), and thus ignore a major source of pi-blocking. Such *suspension-oblivious* (*s-oblivious*) schedulability tests can still be employed to analyze task systems that exhibit self-suspensions, but require that pi-blocking be accounted for pessimistically by inflating each execution requirement e_i by b_i prior to applying the schedulability test. This results in sound, but likely pessimistic results: over-approximating all pi-blocking as additional processor demand is safe because converting execution time to suspensions never increases response times of any task (assuming preemptive JLFP scheduling), but is also likely pessimistic as the processor load is (much) lower in practice than assumed during analysis.

For an example of an s-oblivious schedulability test, consider Liu and

Layland's classic uniprocessor EDF utilization bound for implicit-deadline tasks: a set of *independent* sporadic tasks τ is schedulable under EDF on a uniprocessor if and only if $\sum_{T_i \in \tau} \frac{e_i}{p_i} \leq 1$ [18]. This test is s-oblivious because tasks are assumed to be independent (*i.e.*, there are no shared resources) and because jobs are assumed to always be ready (*i.e.*, there are no self-suspensions). However, even if these assumptions are violated (*i.e.*, if $b_i > 0$ for some T_i), Liu and Layland's utilization bound can still be used as follows [11–13]: in the presence of locking-related self-suspensions, a set of resource-sharing, implicit-deadline sporadic tasks τ is schedulable under EDF on a uniprocessor if

$$\sum_{T_i \in \tau} \frac{e_i + b_i}{p_i} \leq 1. \tag{3}$$

While s-oblivious schedulability analysis may at first sight appear too pessimistic to be useful, it is still relevant because some of the pessimism can actually be "reused" to obtain less pessimistic pi-blocking bounds, and because many published multiprocessor schedulability tests (*e.g.*, [3–8, 16]) do not account for self-suspensions explicitly.

In contrast, effective *suspension-aware* (*s-aware*) schedulability analysis, which explicitly accounts for the effects of pi-blocking, has been developed for select JLFP policies, with *response-time analysis* (RTA) for FP scheduling being a prime example [1, 17]. RTA can be applied to partitioned scheduling (*i.e.*, if $m_j = 1$ for each C_j) as follows. Let b_i^r denote a bound on maximum *remote* pi-blocking (*i.e.*, pi-blocking caused by tasks or agents assigned to remote clusters), and let b_i^l denote a bound on maximum *local* pi-blocking (*i.e.*, pi-blocking caused by tasks or agents assigned to cluster $C(T_i)$), where $b_i = b_i^r + b_i^l$. Then, assuming constrained deadlines (*i.e.*, $d_i \leq p_i$), a task T_i's maximum response time r_i is bounded by the smallest solution to the following recursion [1, 17]:

$$r_i = e_i + b_i^r + b_i^l + \sum_{T_h \in hp(T_i)} \left\lceil \frac{r_i + b_h^r}{p_h} \right\rceil \cdot e_h, \tag{4}$$

where $hp(T_i)$ denotes the set of tasks assigned to processor $C(T_i)$ that have higher priorities than T_i. Equation (4) constitutes an s-aware schedulability test because $b_i = b_i^r + b_i^l$ is explicitly accounted for.

2.3.2 S-Oblivious and S-Aware PI-Blocking

From the point of view of schedulability analysis, a priority inversion exists if a job is delayed (*i.e.*, not scheduled) and this delay *cannot* be attributed

to the execution of a higher-base-priority job (regular interference due to the scheduling of higher-priority jobs are accounted for by any sound schedulability test). Prior work [11–13] has shown that, since s-oblivious schedulability analysis over-approximates a task's processor demand, the definition of "priority inversion" is actually different under s-oblivious and s-aware analysis.

Def. 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 assigned to $C(T_i)$ are **pending**.

Def. 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 assigned to $C(T_i)$ are **scheduled**.

Note that s-oblivious pi-blocking is a weaker definition than s-aware pi-blocking (a scheduled job is always pending, but the reverse is not necessarily true). Therefore, a lower bound on s-oblivious pi-blocking also implies a lower bound on s-aware pi-blocking. Conversely, an upper bound on s-aware pi-blocking implies an upper bound on s-oblivious pi-blocking [11]. We use this relationship in Sec. 3 below.

From a practical point of view, the difference between s-oblivious and s-aware pi-blocking suggests that it is useful to design locking protocols specifically for a particular type of analysis. From an optimality point of view, which we review next, the difference between s-oblivious and s-aware pi-blocking is fundamental because—in shared-memory systems—the two types of analysis have been shown to yield two *different* lower bounds on the amount of pi-blocking that is unavoidable under any locking protocl [11, 13].

2.3.3 PI-Blocking Complexity

The primary purpose of a real-time locking protocol is to minimize pi-blocking to the extent possible. However, if resources are subject to mutual exclusion constraints, then *some* pi-blocking (of either type) is unavoidable in the worst case. A natural question to ask is then: how much is "some," that is, what is the fundamental lower bound on s-aware and s-oblivious pi-blocking that *any* locking protocol can guarantee in the worst case? To enable systematic study of this question, *maximum pi-blocking*, formally $\max\{b_i \mid T_i \in \tau\}$, has been proposed in prior work [11–13] as a metric of blocking complexity.

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. 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 ratios Φ and $\max\{p_i\}/\min\{p_i\}$, the number of resources n_r, or the number of tasks sharing each ℓ_q.

Surprisingly, it was shown [11–13] that, in the case of shared-memory locking protocols, the lower bound on unavoidable pi-blocking depends on whether s-oblivious or s-aware schedulability analysis is employed. More specifically, it was shown that there exist pathological task sets such that maximum pi-blocking is linear in the number of processors m under s-oblivious analysis, but linear in the number of tasks n under s-aware analysis [11–13]. Further, it was shown that these bounds are asymptotically tight with the construction of shared-memory semaphore protocols that ensure for *any* task set maximum pi-blocking that is within a constant factor of the established lower bounds.

To summarize, it has been shown that, in the case of *shared-memory* semaphore protocols, the real-time mutual exclusion problem can be solved with $\max\{b_i \mid T_i \in \tau\} = \Theta(m)$ under s-oblivious schedulability analysis, and $\max\{b_i \mid T_i \in \tau\} = \Theta(n)$ under s-aware schedulability analysis [11–13]. We can now precisely state the contribution of this paper: in the following sections, we establish upper and lower bounds on maximum pi-blocking under s-oblivious and s-aware schedulability analysis for *distributed* real-time locking protocols, thereby complementing the earlier results on shared-memory real-time locking protocols [11–13].

3 Lower Bounds on Maximum PI-Blocking

In this section, we start by establishing a lower bound on maximum s-oblivious and s-aware pi-blocking in a co-hosted setting. We do this by showing the existence of pathological task sets in which some task always incurs $\Omega(\Phi \cdot n)$ pi-blocking, regardless of whether s-oblivious or s-aware schedulability analysis is used. This family of task sets is defined as follows.

Def. 3. For a given smallest cluster size m_1, a given number of tasks n (where $n \geq m \geq 2 \cdot m_1$), and an arbitrary positive integer parameter R (where $R \geq 1$), let $\tau^{seq}(n, m_1, R) \triangleq \{T_1, \ldots, T_n\}$ denote a set of n periodic tasks, with parameters as given in Table 1, that share one resource ℓ_1 local to cluster C_1 (*i.e.*, $C(\ell_1) = C_1$).

The task set $\tau^{seq}(n, m_1, R)$ depends on the smallest cluster size m_1 because, by construction, the maximum pi-blocking will be incurred by tasks in cluster C_1. Note in Table 1 that the maximum critical section lengths

e_i	p_i	d_i	$N_{i,1}$	$L_{i,q}$	$C(T_i)$	
$\frac{R \cdot n}{2}$	$R \cdot n$	$R \cdot n$	0	0	C_1	for $i \in \{1, \ldots, m_1\}$
a	n	$n+1$	1	b	$C_2, \ldots, C_K$	for $i \in \{m_1 + 1, \ldots, 2 \cdot m_1\}$
a	n	$n+1$	1	a	$C_2, \ldots, C_K$	for $i > 2 \cdot m_1$ (if any)

Table 1: Parameters of the tasks in $\tau^{seq}(n, m_1, R)$, where $a = 1$ and $b = 2$ if $m_1 > 1$, and $a = \frac{1}{2}$ and $b = 1$ if $m_1 = 1$. Tasks $T_1, \ldots, T_{m_1}$ are assigned to the first cluster C_1; all other tasks are assigned in a round-robin fashion to clusters other than C_1, *e.g.*, $C(T_i) = C_{(2+i \bmod (K-1))}$. Recall that $n \geq m \geq 2 \cdot m_1$.

(w.r.t. ℓ_1) depend on m_1, which is required to accommodate the special case of $m_1 = 1$. We first consider the case of $m_1 > 1$.

In the following, we assume a synchronous periodic arrival sequence, that is, each task T_i releases a job at time zero and periodically every p_i time units thereafter. We consider periodic tasks (and not sporadic tasks) in this section because it simplifies the example, and since periodic task activation is a special case of sporadic task activation and thus sufficient for establishing a lower bound.

For simplicity, we further assume that each job of tasks $T_{m_1+1}, \ldots, T_n$ immediately accesses resource ℓ_1 as soon as it is allocated a processor (*i.e.*, at the very beginning of the job). This results in a pathological schedule in which tasks $T_{m_1+1}, \ldots, T_n$ are serialized on ℓ_1. An example schedule for $K = 2$, $m_1 = 2$, $m_2 = 2$, $n = 5$, and $R = 3$ is shown in Fig. 1.

We begin by observing that the agent servicing requests for ℓ_1, denoted A_1 in the following, continuously occupies one of the processors in cluster C_1.

Lemma 1. *If $m_1 > 1$, then only $m_1 - 1$ processors are available in cluster C_1 to service jobs of tasks $T_1, \ldots, T_{m_1}$.*

Proof. By construction, the agent A_1 servicing requests for resource ℓ_1 is located in cluster C_1. By Assumption A1, when servicing requests, agent A_1 preempts any job of $T_1, \ldots, T_{m_1}$. By Assumption A2, and since there exists only a single shared resource, agent A_1 becomes active as soon as a request for ℓ_1 is issued. Thus, a processor in C_1 is unavailable for servicing jobs of tasks $T_1, \ldots, T_{m_1}$ whenever A_1 is servicing requests issued by jobs of tasks $T_{m_1+1}, \ldots, T_n$.

Consider an interval $[t_a, t_a + n)$, where $t_a = x \cdot n$ and $x \in \mathbb{N}$. Assuming a synchronous, periodic arrival sequence, tasks $T_{m_1+1}, \ldots, T_n$ each release a

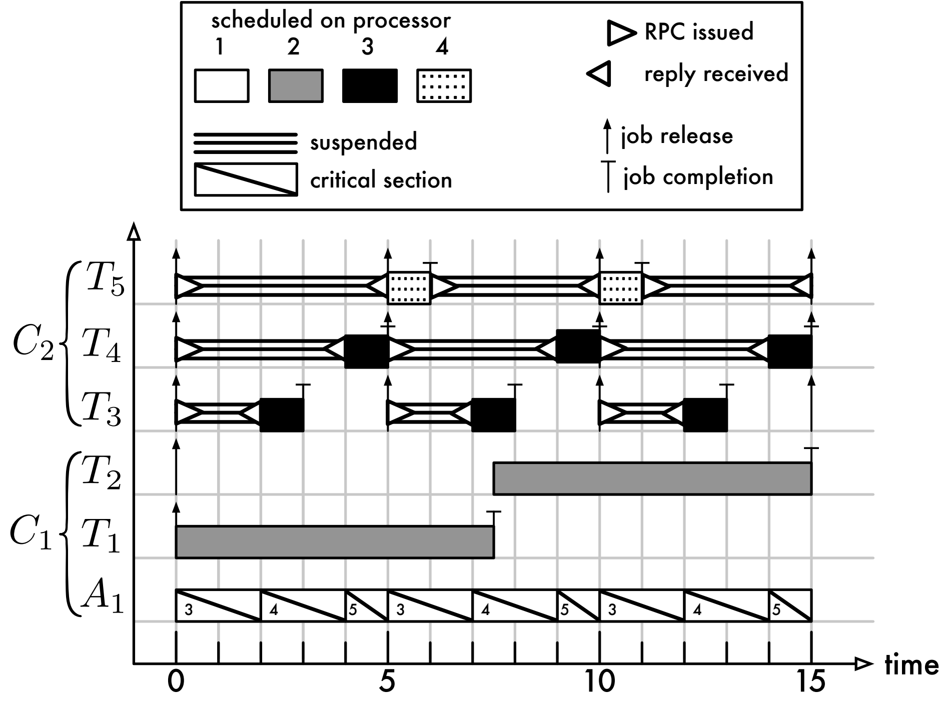

Figure 1: Example schedule of the task set $\tau^{seq}(n, m_1, R)$ as specified in Table 1 for $K = 2$, $m_1 = 2$, $m_2 = 2$, $n = 5$, and $R = 3$. There are five tasks $T_1, \ldots, T_5$ assigned to $K = 2$ clusters sharing one resource ℓ_1, which is local to cluster C_1. Agent A_1 is hence assigned to cluster C_1. The small digit in each critical section signifies the task on behalf of which the agent is executing the request. Deadlines have been omitted from the schedule for the sake of clarity. By construction, the scheduling policy employed to schedule jobs is irrelevant (for simplicity, assume FP scheduling, where lower-indexed tasks have higher priority than higher-indexed tasks). The response-time of T_2 is $r_2 = n \cdot R = 5 \cdot 3 = 15$ since it has the lowest priority in its assigned cluster C_1, and because agent A_1 is continuously occupying a processor.

job at time t_a. Upon being scheduled, each such job immediately accesses resource ℓ_1 and suspends until its request is serviced. As a result, regardless of the JLFP policy used to schedule jobs, A_1 is active during $[t_a, t_a + n)$ for the cumulative duration of all requests issued by jobs of tasks $T_{m_1+1}, \ldots, T_n$ released at time t_a. Assuming each request requires the maximum time to

service, agent A_1 is thus active for a duration of

$$\begin{aligned}\sum_{i=m_1+1}^{n} N_{i,1} \cdot L_{i,1} &= \sum_{i=m_1+1}^{2m_1} N_{i,1} \cdot L_{i,1} + \sum_{i=2m_1+1}^{n} N_{i,1} \cdot L_{i,1} \\ &= \sum_{i=m_1+1}^{2m_1} 2 + \sum_{i=2m_1+1}^{n} 1 \\ &= \sum_{i=1}^{2m_1} 1 + \sum_{i=2m_1+1}^{n} 1 \\ &= n\end{aligned}$$

time units during the interval $[t_a, t_a + n)$, regardless of how the employed locking protocol serializes requests for ℓ_1. Hence, only $m_1 - 1$ processors are available to service jobs of $T_1, \ldots, T_{m_1}$ during the interval $[t_a, t_a + n)$. Since such intervals are contiguous (as $t_a = x \cdot n$ and $x \in \mathbb{N}$), one processor in C_1 is continuously unavailable to jobs of $T_1, \ldots, T_{m_1}$, regardless of either the employed JLFP scheduling policy or the employed locking protocol. □

This in turn implies that the execution of one of the jobs of tasks $T_1, \ldots, T_{m_1}$ is delayed.

Lemma 2. *If* $m_1 > 1$, *then* $\max\{r_i \mid 1 \leq i \leq m_1\} = R \cdot n$.

Proof. Consider an interval $[t_a, t_a + R \cdot n)$, where $t_a = x \cdot R \cdot n$ and $x \in \mathbb{N}$. Assuming a synchronous, periodic arrival sequence, tasks $T_1, \ldots, T_{m_1}$ each release a job at time t_a. Regardless of the (work-conserving) JLFP policy employed to assign priorities to jobs, one of these m_1 jobs will have lower priority than the other $m_1 - 1$ ready pending jobs in cluster C_1. Let J_l denote this lowest-priority job. By Lemma 1, there are only $m_1 - 1$ processors available to service jobs. Thus J_l will only be scheduled after one of the other jobs has finished execution. Since each task assigned to cluster C_1 has a worst-case execution time of $e_i = \frac{R \cdot n}{2}$, in the worst case, job J_l is not scheduled until time $t_a + \frac{R \cdot n}{2}$, and then requires another $e_l = \frac{R \cdot n}{2}$ time units of processor service to complete. Hence, $\max\{r_i \mid 1 \leq i \leq m_1\} = 2e_i = R \cdot n$. □

So far we have considered only the case of $m_1 > 1$. By construction, the same maximum response-time bound arises also in the case of $m_1 = 1$.

Lemma 3. *If* $m_1 = 1$, *then* $\max\{r_i \mid 1 \leq i \leq m_1\} = R \cdot n$.

Proof. If $m_1 = 1$, then there is only one task assigned to cluster C_1. The single processor in C_1 is available to jobs of T_1 only when A_1 is inactive. Recall from Table 1 that the maximum critical section lengths of tasks

$T_{m_1+1}, \ldots, T_n$ are halved if $m_1 = 1$. Analogously to Lemma 1, it can thus be shown that, in the worst case, the single processor in C_1 is available to T_1 for only $\frac{n}{2}$ time units out of each interval $[x \cdot n, x \cdot n + n)$, where $x \in \mathbb{N}$.

Consider an interval $[t_a, t_a + R \cdot n)$, where $t_a = x \cdot R \cdot n$ and $x \in \mathbb{N}$. Assuming a synchronous arrival sequence, task T_1 releases a J_1 at time t_a. In the worst case, J_1 requires $e_1 = \frac{R \cdot n}{2}$ time units to complete. Assuming maximum interference by A_1 (*i.e.*, if the processor is unavailable to J_1 for $\frac{n}{2}$ time units every n time units), J_1 will accumulate e_1 time units of processor service only by time $t_a + 2e_i = t_a + R \cdot n$. □

Since there are m_1 processors and m_1 pending jobs in cluster C_1, this implies the existence of s-oblivious pi-blocking.

Lemma 4. *Under any* JLFP *scheduling policy and s-oblivious schedulability analysis,* $\max\{b_i \mid T_i \in \tau^{seq}(n, m_1, R)\} \geq \frac{R \cdot n}{2}$.

Proof. By construction, there are at most m_1 pending jobs in cluster C_1 at any time. This implies that any delay of a pending job constitutes s-oblivious pi-blocking (recall Definition 1), that is, $b_i = r_i - e_i$ for each $T_i \in \{T_1, \ldots, T_{m_1}\}$, regardless of the employed JLFP scheduling policy. Since $e_i = \frac{R \cdot n}{2}$ for each $T_i \in \{T_1, \ldots, T_{m_1}\}$, we have $\max\{b_i \mid 1 \leq i \leq n\} \geq \max\{r_i \mid 1 \leq i \leq m_1\} - \frac{R \cdot n}{2}$. By Lemmas 2 and 3, $\max\{r_i \mid 1 \leq i \leq m_1\} = R \cdot n$, and thus $\max\{b_i \mid 1 \leq i \leq n\} \geq R \cdot n - \frac{R \cdot n}{2} = \frac{R \cdot n}{2}$. □

Since the agent A_1 and tasks $T_1, \ldots, T_{m_1}$ share a cluster in $\tau^{seq}(n, m_1, R)$, and because the s-oblivious pi-blocking implies s-aware pi-blocking, we obtain the following lower bound on maximum pi-blocking in a co-hosted setting.

Theorem 1. *Under* JLFP *scheduling, using either s-aware or s-oblivious schedulability analysis, there exists a task set such that maximum pi-blocking* $\max\{b_i\} = \Omega(\Phi \cdot n)$ *in a co-hosted setting under any weakly work-conserving distributed multiprocessor real-time semaphore protocol employing priority-boosted agents (i.e., under protocols matching Assumptions A1 and A2).*

Proof. By Lemma 4, for any $R \in \mathbb{N}$, there exists a task set $\tau^{seq}(n, m_1, R)$ such that $\max\{b_i \mid T_i \in \tau^{seq}(n, m_1, R)\} \geq \frac{R \cdot n}{2}$ under s-oblivious schedulability analysis for any JLFP policy and any distributed multiprocessor semaphore protocol satisfying Assumptions A1 and A2. Recall from Equation (1) that $\Phi = \frac{\max\{r_i\}}{\min\{p_i\}}$, and hence $\Phi = \frac{R \cdot n}{n} = R$ in the case of $\tau^{seq}(n, m_1, R)$. Since R can be freely chosen, we have $\max\{b_i\} = \Omega(R \cdot n) = \Omega(\Phi \cdot n)$ under s-oblivious schedulability analysis. The presented analysis also extends to s-aware schedulability analysis since Definition 1 is weaker than Definition 2; lower-bounds on s-oblivious pi-blocking thus also imply lower bounds on s-aware pi-blocking [11]. □

In the s-aware case, Theorem 1 shows that in the distributed setup with co-hosted tasks and agents, maximum pi-blocking is *asymptotically worse* by a factor of Φ, compared to a shared-memory system, where the equivalent mutual exclusion problem can be solved with $\Theta(n)$ maximum s-aware pi-blocking in the general case [11, 13]. Maximum s-oblivious pi-blocking is also asymptotically worse—the equivalent mutual exclusion problem can be solved with $\Theta(m)$ maximum s-oblivious pi-blocking[11–13] (recall that we assume $n \geq m$). Crucially, Φ, the ratio of the maximum response time and the minimum period, can be chosen to be arbitrarily large and is independent of either m or n. This suggests that, from a schedulability point of view, the mutual exclusion problem is fundamentally more difficult to handle in a distributed environment.

The observed discrepancy, however, is entirely due to the effects of *preemptions* due to priority-boosted agents. While it is necessary to boost the priority of agents to ensure a timely completion of requests, such troublesome preemptions can be ruled out entirely by disallowing the co-hosting of agents and tasks in the same cluster. In fact, in such a scenario with dedicated synchronization processors (or clusters), the asymptotic lower bounds in the distributed case are the same as in an equivalent shared-memory setting. These lower bounds can be trivially established with the setup previously used in [11]; we omit the details here and summarize the correspondence with the following theorem.

Theorem 2. *Under JLFP scheduling, there exist task sets such that, in a disjoint setting, under any distributed real-time semaphore protocol matching Assumptions A1 and A2,* $\max\{b_i\} = \Omega(n)$ *under s-aware schedulability analysis and* $\max\{b_i\} = \Omega(m)$ *under s-oblivious schedulability analysis.*

In other words, the previously established lower bounds on maximum pi-blocking under *any* shared-memory locking protocol apply to the distributed case as well since distributed locking protocols can also be implemented in a shared-memory system. In Secs. 5.2 and 5.1, we show these bounds on maximum pi-blocking to be asymptotically tight (in a disjoint setting) with the construction of two asymptotically optimal distributed real-time semaphore protocols.

As a final remark, we note that the task set $\tau^{seq}(n, m_1, R)$ as given in Table 1 contains tasks with relative deadlines larger than their periods (*i.e.*, $d_i > p_i$ for $i > m_1$). This is purely a matter of convenience and asymptotically unchanged bounds can easily be derived with implicit-deadline tasks only.

Having established a lower bound of $\Omega(\Phi \cdot n)$, we next explore the question of asymptotic optimality in co-hosted settings.

4 Asymptotic Optimality in a Co-Hosted Setting

Intuitively, Theorem 1 shows that there exist pathological scenarios in which the choice of real-time locking protocol is seemingly irrelevant. That is, regardless of the specifics of the employed locking protocol, worst-case pi-blocking appears to be asymptotically worse than in a comparable shared-memory system simply because resources are inaccessible from some processors. Curiously, protocol-specific rules *are* immaterial from an asymptotic point of view: as established next, *any* distributed real-time locking protocol that does not starve requests is *asymptotically optimal* if co-hosting is allowed.

Theorem 3. *Under any JLFP scheduler, any weakly-work-conserving, distributed real-time semaphore protocol that employs priority boosting (i.e., any protocol matching Assumptions A1 and A2) ensures $O(\Phi \cdot n)$ maximum pi-blocking, regardless of whether s-aware or s-oblivious schedulability analysis is employed.*

Proof. A pi-blocked job J_b incurs *s-aware* pi-blocking either when **(i)** it is suspended while waiting for a resource request to be completed, or when **(ii)** it is preempted while a local priority-boosted agent completes a request for another job.

Concerning (i), the completion of J_b's own requests can only be delayed by other requests (and not by the execution of other jobs) since agents are priority-boosted, and since the employed distributed locking protocol is weakly work-conserving (*i.e.*, whenever one of J_b's requests is delayed, at least one other request is being processed by some agent).

Concerning (ii), agents only become active when invoked by other jobs.

Hence the total duration of all requests (issued by jobs of *any* task) that are executed while J_b is pending provides an upper bound on the maximum duration of pi-blocking incurred by J_b. Since each job is assumed to issue at most a constant number of requests of constant length (*i.e.*, from an asymptotic blocking complexity point of view, for all T_i, $\sum_{\ell_q} N_{i,q} \cdot L_{i,q} = O(1)$ [11, 13]), the maximum number of jobs that may be scheduled while J_b is pending yields a suitable upper bound.

For any task T_x, the number of jobs of T_x that execute while J_b is pending is bounded by $\left\lceil \frac{r_x + r_b}{p_x} \right\rceil$.[1] Since there are n tasks in total, this implies that at

[1]See *e.g.* [13, Ch. 4] for a formal proof of this well-known bound.

most

$$\sum_{x=1}^{n} \left\lceil \frac{r_x + r_b}{p_x} \right\rceil = \sum_{x=1}^{n} \left\lceil \frac{r_x}{p_x} + \frac{r_b}{p_x} \right\rceil$$
$$\leq \sum_{x=1}^{n} \left\lceil \frac{\max_i\{r_i\}}{\min_i\{p_i\}} + \frac{\max_i\{r_i\}}{\min_i\{p_i\}} \right\rceil$$
$$= n \cdot \lceil 2\Phi \rceil = O(\Phi \cdot n)$$

jobs are executed while J_b is pending. Since $\sum_{\ell_q} N_{i,q} \cdot L_{i,q} = O(1)$ for each T_i, it follows that $\max_i\{b_i\} = O(n \cdot \Phi)$, regardless of any protocol-specific rules. The presented analysis also extends to s-oblivious schedulability analysis since Definition 1 is weaker than Definition 2; upper-bounds on s-aware pi-blocking thus also imply upper bounds on s-aware pi-blocking [11]. □

As a corollary, Theorem 3 implies that the DPCP, which orders requests according to task priority, is asymptotically optimal in the co-hosted setting. However, it also shows that requests may be processed in arbitrary order (*e.g.*, in FIFO order, or even in random order) without losing asymptotic optimality (as long as at least one request at a time is satisfied and agents are priority-boosted), which is surprising given that the choice of queue order is crucial in the shared-memory case [11].

As already noted, in the previous section, by prohibiting the co-hosting of resources and tasks—that is, somewhat counter-intuitively, by making the system *less* similar to a shared-memory system (in which resources and tasks are necessarily co-hosted)—it is indeed possible to ensure maximum s-aware pi-blocking that is asymptotically no worse than in a shared-memory system. Not surprisingly, the choice of queue order is significant in this case. We next introduce suitable protocols that realize asymptotically optimal maximum pi-blocking under s-aware and s-oblivious schedulability analysis.

5 Asymptotic Optimality in a Disjoint Setting

Prior work [9, 11–13] has established shared-memory protocols that yield upper bounds on maximum s-aware and s-oblivious pi-blocking of $O(n)$ and $O(m)$, respectively. These protocols, namely the *FIFO Multiprocessor Locking Protocol* (FMLP$^+$) for s-aware analysis [9, 13] and the family of $O(m)$ *Locking Protocols* (the OMLP family) for s-oblivious analysis [11–13], rely on specific queue structures with strong progress guarantees to obtain the desired bounds. In the following, we show how the key ideas underlying the FMLP$^+$ and the OMLP family can be adopted to the problem of designing

asymptotically optimal locking protocols for the distributed case studied in this paper. We begin with the slightly simpler s-aware case.

5.1 Optimal Maximum S-Aware PI-Blocking

Inspired by the FMLP$^+$ [13], the *Distributed FIFO Locking Protocol* (DFLP) relies on simple FIFO queues to avoid starvation. Notably, the DFLP ensures $O(n)$ maximum s-aware pi-blocking in a disjoint setting and transparently supports arbitrary, non-uniform cluster sizes (*i.e.*, unlike the DPCP, the DFLP supports distributed systems consisting of multiprocessor nodes with $m_j > 1$ for some C_j and allows $m_j \neq m_h$ for $j \neq h$). We first describe the structure and rules of the DFLP, and then establish its asymptotic optimality.

Rules. Under the DFLP, conflicting requests for each serially-reusable resource ℓ_q are ordered with a per-resource FIFO queue FQ_q. Requests for ℓ_q are served by an agent A_q assigned to ℓ_q's cluster $C(\ell_q)$. Resource requests are processed according to the following rules.

1. When J_i issues a request $\mathcal{R}$ for resource ℓ_q, J_i suspends and $\mathcal{R}$ is appended to FQ_q. J_i's request is processed by agent A_q when $\mathcal{R}$ becomes the head of FQ_q.

2. When $\mathcal{R}$ is complete, it is removed from FQ_q and J_i is resumed.

3. Agent A_q is inactive when ℓ_q's request queue FQ_q is empty and active when it is processing requests. Active agents are scheduled preemptively in order of increasing issue times with regard to each agent's currently-processed request (*i.e.*, an agent processing an earlier-issued request has higher priority than one serving a later-issued request). Any ties in request times can be broken arbitrarily (*e.g.*, in favor of agents serving requests of lower-indexed tasks).

4. Agents have statically higher priority than jobs (*i.e.*, agents are subject to priority-boosting).

We next show that these simple rules yield asymptotic optimality.

Blocking complexity. The co-hosted case is trivial since the DFLP uses priority boosting (Rule 4) and because it is weakly work-conserving (requests are satisfied as soon as the requested resource is available—see Rule 1); Theorem 3 hence applies.

To show that the DFLP is asymptotically optimal in the disjoint case as well, we first establish a per-request bound on the number of interfering requests that derives from FIFO-ordering both requests and agents.

Lemma 5. *Let $\mathcal{R}$ denote a request issued by a job J_i for a resource ℓ_q and let T_x denote a task ($i \neq x$). Under the DFLP, jobs of T_x delay the completion of $\mathcal{R}$ with at most one request.*

Proof. J_i's request $\mathcal{R}$ cannot be delayed by later-issued requests since FQ_q is FIFO-ordered and because agents are scheduled in FIFO order according to the issue time of the currently-served request. Suppose $\mathcal{R}$ is delayed by two or more requests issued by jobs of T_x. Since $\mathcal{R}$ is not delayed by later-issued requests (and clearly not by earlier-completed requests), all blocking requests are incomplete at the time that $\mathcal{R}$ is issued. Since tasks and jobs are both sequential, and since jobs request at most one resource at a time, there exists at most one incomplete request per task at any time. Contradiction. □

An $O(n)$ bound on maximum pi-blocking follows immediately since each of the other $n - 1$ tasks delays J_i at most once each time J_i requests a resource, and since agents do not preempt jobs in the disjoint setting.

Theorem 4. *In a disjoint setting, the DFLP ensures $O(n)$ maximum s-aware pi-blocking.*

Proof. Let J_i denote an arbitrary job. Since, by assumption, no agents execute on J_i's cluster, J_i incurs pi-blocking only when suspended while waiting for a request to complete. By Lemma 5, each other task delays each of J_i's $\sum_q N_{i,q}$ requests at most for the duration of one request, that is, *per request*, J_i incurs no more than $n \cdot L^{max}$ s-aware pi-blocking. Hence, since J_i issues at most $\sum_q N_{i,q}$ requests, we have $b_i \leq n \cdot L^{max} \cdot \sum_q N_{i,q} = O(n)$ since $\sum_q N_{i,q}$ and L^{max} are assumed to be constants. □

The DFLP is thus asymptotically optimal with regard to maximum s-aware pi-blocking, in both co-hosted settings (Theorem 3) and in settings with dedicated synchronization processors (Theorem 4). In contrast, the DPCP does not generally guarantee $O(n)$ s-aware pi-blocking even if dedicated synchronization processors are employed since it orders conflicting requests by task priority and is thus prone to starvation issues (this can be shown similar to the lower bound on priority queues established in [11, 13]).

Next, we consider the s-oblivious case.

5.2 Optimal Maximum S-Oblivious PI-Blocking

In this section, we define and analyze the *Distributed $O(m)$ Locking Protocol* (D-OMLP), which augments the OMLP family with support for the distributed system model.

In order to prove optimality under s-oblivious analysis, a protocol must ensure an upper bound of $O(m)$ s-oblivious pi-blocking. Since there are $n \geq m$ tasks in total, if each task is allowed to submit a request concurrently, excessive contention could arise at each agent (*i.e.*, if an agent is faced with n concurrent requests, regardless of the order in which requests are processed, it is not possible to ensure $O(m)$ maximum s-oblivious pi-blocking). Thus, it is necessary to limit contention early within each application cluster (where job priorities can be taken into account) to only allow a subset of high-priority jobs to invoke agents at the same time. In the interest of practicality, such "contention limiting" should not require coordination across clusters, but rather must be based only on local information. As we show next, this can be accomplished by reusing (aspects of) two protocols of the OMLP family.

The first technique is to introduce *contention tokens*, which are virtual, cluster-local resources that a job must acquire prior to invoking an agent. This technique was previously used in the shared-memory OMLP variant for partitioned JLFP scheduling [11]. By limiting the number of contention tokens to m in total (*i.e.*, by assigning exactly m_j such tokens to each cluster C_j), each agent is faced with at most m concurrent requests.

This in turn creates the problem of managing access to contention tokens. However, since contention tokens are a cluster-local resource, this reduces to a shared-memory problem and prior results on optimal shared-memory real-time synchronization can be reused. In fact, as there may be multiple contention tokens in each cluster (if $m_j > 1$), of which a job may use any one, this reduces to a k-exclusion problem (where k denotes the number of tokens per cluster in this case). Several asymptotically optimal solutions for the k-exclusion problem under s-oblivious analysis have been developed [12, 15, 25], including a variant of the OMLP [12]; the contention tokens can thus be readily managed within each cluster using one of the available k-exclusion protocols. These considerations lead to the following protocol definition.

Rules. Under the D-OMLP, there are m_j contention tokens in each cluster C_j, for a total of $m = \sum_{j=1}^{K} m_j$ contention tokens in total. As in the DFLP, there is one agent A_q and a FIFO queue FQ_q for each resource ℓ_q.

Jobs may access shared resources according to the following rules. In the

following, let J_i denote a job that must access resource ℓ_q.

1. Before J_i may invoke agent A_q, it must first acquire a contention token local to cluster $C(T_i)$ according to the rules of an asymptotically optimal k-exclusion protocol (*e.g.*, any of the protocols from [12, 15, 25]).

2. Once J_i holds a contention token, it immediately issues its request $\mathcal{R}$ by invoking A_q and suspends. $\mathcal{R}$ is appended to FQ_q, and processed by A_q when it becomes the head of FQ_q.

3. When $\mathcal{R}$ is complete, it is removed from FQ_q. J_i is resumed and immediately relinquishes its contention token.

4. As in the DFLP, agent A_q is active whenever FQ_q is non-empty, and inactive otherwise. Active, ready agents are scheduled preemptively in order of non-decreasing request enqueue times (*i.e.*, while processing $\mathcal{R}$, agent A_q's priority is the point in time at which $\mathcal{R}$ was enqueued in FQ_q). Any ties in the times that requests were enqueued can be broken arbitrarily.

5. Agents have a statically higher priority than jobs (*i.e.*, agents are subject to priority-boosting).

Processing requests in FIFO order, together with the contention token abstraction, yields asymptotically optimal maximum s-oblivious pi-blocking, as we show next.

Blocking complexity. As with the DFLP, the co-hosted case is trivial since Theorem 3 applies to the D-OMLP.

In the disjoint case, we first establish a bound on the *maximum token-hold time*, since jobs can incur s-oblivious pi-blocking both due to Rule 1 (*i.e.*, when no contention tokens are available) and due to Rules 2 and 4 (*i.e.*, when $\mathcal{R}$ is preceded by other requests in FQ_q or if A_q is preempted while processing $\mathcal{R}$).

Lemma 6. *In a setting with dedicated synchronization clusters, a job J_i holds a contention token for at most $m \cdot L^{max}$ time units per request.*

Proof. By Rules 1 and 3, a job J_i holds a contention token while it waits for its request $\mathcal{R}$ to be completed. Analogously to Lemma 5, since FQ_q is FIFO-ordered, and since agents are scheduled FIFO order w.r.t. the time that requests are enqueued (Rule 4), the completion of $\mathcal{R}$ can only be delayed due to the execution of requests that were incomplete at the time that $\mathcal{R}$ was

enqueued in FQ_q. By Rule 1, only jobs holding a contention token may issue requests to agents. Since there are only $m = \sum_{j=1}^{K} m_j$ contention tokens in total, there exist at most $m - 1$ incomplete requests at the time that $\mathcal{R}$ is enqueued in FQ_q. Hence, $\mathcal{R}$ is completed and J_i relinquishes its contention token after at most $m \cdot L^{max}$ time units. □

By leveraging prior asymptotically optimal k-exclusion locking protocols for s-oblivious analysis, Lemma 6 immediately yields an $O(m)$ bound on maximum s-oblivious pi-blocking under the D-OMLP.

Theorem 5. *In a disjoint setting, the D-OMLP ensures $O(m)$ maximum s-oblivious pi-blocking.*

Proof. Let H denote the maximum token-hold time. By Lemma 6, the maximum token-hold time is $H = m \cdot L^{max} = O(m)$. Further, H represents the "maximum critical section length" w.r.t. the contention token k-exclusion problem. By Rule 1, an asymptotically optimal k-exclusion protocol is employed to manage access to contention tokens within each cluster. Applied to a cluster with m_j processors, the k-exclusion problem can be solved such that jobs incur s-oblivious pi-blocking for the duration of at most $O\left(\frac{m_j}{k}\right)$ critical section lengths per request [12, 13]. By definition of the D-OMLP, there are $k = m_j$ contention tokens in each cluster C_j. Hence, in a disjoint setting, a task assigned to a cluster C_j incurs at most $O\left(\frac{m_j}{m_j} \cdot H\right) = O(H) = O(m)$ s-oblivious pi-blocking. □

The D-OMLP is thus asymptotically optimal under s-oblivious schedulability analysis, and hence a natural extension of the OMLP family to the distributed real-time locking problem.

6 Conclusion

In this paper, we considered distributed real-time locking protocols that ensure mutual exclusion from an optimality point of view: asymptotically speaking, what is the "best" upper bound on pi-blocking that any such protocol can ensure in the general case? We identified two different task and resource allocation strategies, co-hosted and disjoint settings, that give rise to different answers to this question. In the co-hosted setting, $\Omega(\Phi \cdot n)$ maximum s-aware pi-blocking is unavoidable in pathological cases (under either s-aware or s-oblivious analysis), whereas in the disjoint setting, $\Omega(n)$ maximum s-aware and $\Omega(m)$ maximum s-oblivious pi-blocking is fundamental. We further showed that any distributed locking protocol satisfying Assumptions A1 and A2 is asymptotically optimal in the co-hosted case. For the disjoint case,

we introduced two new distributed real-time semaphore protocols that show the established lower bounds to be asymptotically tight. Specifically, the DFLP is asymptotically optimal under s-aware analysis, and the D-OMLP is asymptotically optimal under s-oblivious analysis, w.r.t. blocking complexity as determined by the maximum pi-blocking metric.

It is important to note that asymptotic optimality does *not* imply that such a protocol is always preferable to a non-optimal one. Rather, blocking bounds of asymptotically similar locking protocols may differ greatly in absolute terms. Whether a particular locking protocol is suitable for a particular task set depends on both the task set's specific requirements and a protocol's constant factors, which asymptotic analysis does not reflect. In particular, this is the case in the co-hosted setting, where all distributed locking protocols (in the considered class of protocols) differ *only* in terms of constant factors. Fine-grained (*i.e.*, non-asymptotic) bounds on maximum pi-blocking suitable for schedulability analysis are thus required for practical use and to enable a detailed comparison. Such bounds should not only reflect a detailed analysis of protocol rules, but also exploit task-set-specific properties such as per-task bounds on request lengths and frequencies. We have recently developed such bounds for the DFLP and the DPCP [10]; the same techniques could also be applied to analyze the D-OMLP.

As already noted in Sec. 2.2, following Rajkumar *et al*. [20, 21], we have made the assumption that jobs can invoke agents with "negligible" overheads (*i.e.*, with overheads that can be accounted for using known overhead accounting techniques [13]). This is a reasonable assumption if distributed semaphore protocols are implemented on top of a (large) shared-memory platform (*e.g.*, see [10] for such a case), but it may be more problematic in systems that require explicit message routing among a shared interconnect. Assuming there exists an upper bound on the message delay between a task T_i and each agent A_q, denoted $\Delta_{i,q}$, such delays can be incorporated simply by increasing T_i's self-suspension time by $2\Delta_{i,q}$ for each agent invocation (under the D-OMLP, the maximum token-hold time is increased by $2\Delta_{i,q}$ as well). If $\Delta_{i,q}$ can be considered constant (*i.e.*, if $\Delta_{i,q} = O(1)$ from an asymptotic analysis point of view), then the asymptotic upper and lower bounds established in this paper remain unaffected. If, however, $\Delta_{i,q}$ may depend on m or n, additional analysis is required.

In another opportunity for future work, it will also be interesting to explore how to accommodate *nested requests*, that is, how to allow complex requests that require agents to invoke other agents. Ward and Anderson have recently shown that arbitrarily deep nesting can be supported in shared-memory locking protocols without loss of asymptotic optimality [24]; however,

it remains to be seen whether their techniques can be extended to the case of distributed real-time semaphore protocols as well.

References

[1] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A.J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

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

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

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

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

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

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

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

[9] B. Brandenburg. An asymptotically optimal real-time locking protocol for clustered scheduling under suspension-aware analysis. In *Proceedings of the 33rd IEEE Real-Time Systems Symposium Work-in-Progress Session*, page 8, 2012.

[10] B. Brandenburg. Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling. In *Proceedings of the 19th*

IEEE Real-Time and Embedded Technology and Applications Symposium, pages 141–152, 2013.

[11] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of the 31st Real-Time Systems Symposium*, pages 49–60, 2010.

[12] B. Brandenburg and J. Anderson. The OMLP family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Systems*, online first, DOI 10.1007/s10617-012-9090-1, July 2012.

[13] Björn Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[14] 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 Hall/CRC, 2004.

[15] G. Elliott and J. Anderson. 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*, pages 15–24, 2011.

[16] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2-3):187–205, 2003.

[17] K. Lakshmanan, D. Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 469–478, 2009.

[18] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 30:46–61, 1973.

[19] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 116–123, 1990.

[20] R. Rajkumar. *Synchronization In Real-Time Systems—A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[21] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. *Proceedings of the 9th IEEE Real-Time Systems Symposium*, pages 259–269, 1988.

[22] F. Ridouard, P. Richard, and F. Cottet. Negative results for scheduling independent hard real-time tasks with self-suspensions. In *Proceedings of the 25th IEEE Real-Time Systems Symposium*, pages 47–56, 2004.

[23] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[24] B. Ward and J. Anderson. Supporting nested locking in multiprocessor real-time systems. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 223–232, 2012.

[25] B. Ward, G. Elliott, and J. Anderson. Replica-request priority donation: A real-time progress mechanism for global locking protocols. In *Proceedings of the 18th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 280–289, 2012.