Blocking Optimality in Distributed Real-Time Locking Protocols

Björn B. Brandenburg

Max Planck Institute for Software Systems (MPI-SWS)
Paul-Ehrlich-Straße G 26, 67663 Kaiserslautern, Germany
bbb@mpi-sws.org

Abstract

Lower and upper bounds on the maximum priority inversion blocking (pi-blocking) that is generally unavoidable in distributed multiprocessor real-time locking protocols (where resources may be accessed only from specific synchronization processors) are established. Prior work on suspension-based shared-memory multiprocessor locking protocols (which require resources to be accessible from all processors) has established asymptotically tight bounds of $\Omega(m)$ and $\Omega(n)$ 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 (i.e., processors hosting critical sections), $\Omega(\Phi \cdot n)$ maximum pi-blocking is unavoidable for some tasks under any locking protocol under both suspension-aware and suspension-oblivious schedulability analysis, where $\Phi$ denotes the ratio of the maximum response time to the shortest period. In contrast, in the case of disjoint task allocation (i.e., if application tasks may not be assigned to synchronization processors), only $\Omega(m)$ and $\Omega(n)$ maximum pi-blocking is fundamentally unavoidable 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 locking protocols that ensure $O(m)$ and $O(n)$ maximum pi-blocking under suspension-oblivious and suspension-aware analysis, respectively.

1 Introduction

The principal purpose of a real-time locking protocol is to provide tasks with mutually exclusive access to shared resources such that the maximum blocking incurred by any task can be bounded a priori. Such blocking is problematic in real-time systems and must be bounded because it increases worst-case response times, and hence may cause deadline violations if left unchecked. Real-time locking protocols should thus avoid blocking as much as possible. Unfortunately, if tasks require exclusive access, some blocking is inherently possible and can generally not be avoided. This naturally raises the question of optimality: if some blocking is inevitable when using locks, then what is the minimal bound on worst-case blocking that any locking protocol can guarantee? In other words, when can a real-time locking protocol be deemed (asymptotically) optimal?

This question has long been answered for uniprocessor systems [2, 44, 48], where it has been shown that the real-time mutual exclusion problem can be solved with $O(1)$ maximum blocking: the priority ceiling protocol (PCP) [44, 48] and the stack resource policy (SRP) [2] both ensure
that the maximum blocking incurred by any task is bounded by the length of a single (outermost) critical section, which is obviously optimal.

In the multiprocessor case, the picture is not as straightforward, and not as complete. First of all, there are two classes of multiprocessor locking protocols to consider: spin-based (or spin lock) protocols, in which waiting tasks remain scheduled and execute a delay loop, and suspension-based (or semaphore) protocols, in which waiting tasks suspend to make the processor available to other tasks. Of the two classes, spin locks are much simpler to analyze: with non-preemptive FIFO spin locks, a lock acquisition is delayed by at most one critical section on each other processor [23, 29], and it is easy to see that this cannot be improved upon in the general case.

In the case of multiprocessor real-time semaphore protocols, however, the question of blocking optimality is considerably more challenging, and has only recently been answered in part [11, 16, 18, 49]. In particular, it has been answered only for the case of shared-memory multiprocessor semaphore protocols, which fundamentally require shared resources to be accessible from all processors because they assume that tasks execute critical sections locally on the processor(s) on which they are scheduled. In this paper, we extend the theory of blocking optimality to distributed multiprocessor semaphore protocols, which are required if (some) shared resource(s) can be accessed only from specific (subsets of) processors.

1.1 Motivation

Besides the fact that the restriction to shared-memory systems in prior work is an obvious limitation, our work is motivated by the observation that there are many systems that either inherently require, or at least can benefit from, distributed real-time locking protocols.

For instance, in the absence of a shared memory or on heterogeneous hardware platforms (e.g., if only some processor cores support special-purpose instructions), the execution of critical sections can be inherently restricted to specific processors. Similarly, when tasks share physical resources such as network links, I/O co-processors, graphics processing units (GPUs), or digital signal processors (DSPs), certain devices may be accessible only from specific processors.

Second, even if all processors technically could access all shared resources, it sometimes is preferable to centralize resource access nonetheless. For example, many shared-memory multicore processors intended for embedded systems are not cache-consistent (e.g., Infineon’s Aurix platform for automotive applications does not support hardware-based cache coherency). On such a platform, the coherency of shared data structures either must be managed in software (thus introducing an additional implementation burden), or alternatively the execution of critical sections can simply be centralized on a dedicated processor with the help of a distributed real-time locking protocol. In fact, even on a cache-consistent shared-memory platform, it can be beneficial to centralize the execution of critical sections due to cache affinity issues [39]. Furthermore, the use of distributed real-time locking protocols in shared-memory systems can also yield improved schedulability [14].

As the final example, consider multi-kernel operating systems [6, 51], where each core is managed as a uniprocessor and system-wide resource management is carried out using message passing. Multi-kernels tend to aggressively optimize locality—intuitively, they form a “distributed system on a chip”—with the effect that some resources may be accessed only on specific cores.

In each of these examples, the critical sections of some tasks must be executed on a specific remote processor, since executing them locally is either infeasible or disallowed. This renders shared-memory semaphore protocols as studied in [11, 16, 18, 49] inapplicable, and a distributed real-time semaphore protocol must be employed instead.

Naturally, as in the uniprocessor and shared-memory cases, distributed real-time locking protocols should minimize blocking to the extent possible. However, to the best of our knowledge, blocking optimality in distributed real-time locking protocols has not been studied to date, and it
is thus not even clear what the minimal “extent possible” is, nor is it known how protocols should be structured to obtain (asymptotically) optimal blocking bounds. In this paper, we seek to close this gap in the understanding of multiprocessor real-time synchronization.

1.2 Related Work

The first discussion of the effects of uncontrolled blocking in real-time systems and possible solutions dates back to the Mesa project [36]. Sha et al. [48] were the first to study the problem from an analytical point of view and proposed uniprocessor protocols that provably bound the worst-case blocking duration. As already mentioned, the Sha et al.'s PCP [48] and Baker’s SRP [2] were the first uniprocessor semaphore protocols to ensure optimal blocking bounds.

In the first work on synchronization in multiprocessor real-time systems, Rajkumar et al. [45] proposed the distributed priority ceiling protocol (DPCP) [44,45] for partitioned\(^1\) multiprocessors, which applies the PCP on each processor and uses “agents” to carry-out critical sections on behalf of tasks assigned to remote processors. As the first and prototypical distributed real-time semaphore protocol, the DPCP is central to this paper and reviewed in greater detail in Section 2.2.2. Rajkumar also developed the first suspension-based shared-memory real-time locking protocol, namely the multiprocessor priority ceiling protocol (MPCP) [43],\(^2\) an extension of the PCP for partitioned shared-memory multiprocessors based on priority queues. Like the DPCP, the MPCP uses the regular PCP for local resources (i.e., resources used on only one processor), but when accessing global resources (i.e., resources used by tasks on multiple processors), tasks execute critical section on their assigned processor in the MPCP (rather than delegating resource access to “agents” as in the DPCP). In contrast to the PCP and the SRP, which are obviously optimal on a uniprocessor, the MPCP and the DPCP were not studied from a blocking optimality perspective.

Favoring spin locks over semaphores, Gai et al. [28,29] developed the MSRP, a multiprocessor extension of the SRP for partitioned shared-memory multiprocessors, which employs non-preemptive FIFO spin locks for global resources and the SRP for local resources; Devi et al. [23] similarly analyzed non-preemptive FIFO spin locks in the context of globally scheduled multiprocessors.\(^3\) As already pointed out, it is not possible to construct spin lock protocols that ensure, in the worst case, asymptotically less blocking to all tasks than the protocols by Gai et al. [28,29] and Devi et al. [23], although it is possible to use priority-ordered spin locks [32,33,41] to ensure that some tasks are less susceptible to blocking than others [52].

In subsequent work on shared-memory real-time locking protocols (both spin-based and suspension-based), numerous new protocols, analysis improvements, and evaluations have been presented [10,11,14,17,19,20,22,24,26,27,35,40,42,47]; however, in contrast to this paper, they are not primarily concerned with questions of blocking optimality.

Perhaps more closely related are two studies targeting different notions of optimality. Soon after the MPCP was proposed, Lortz and Shin [38] observed that ordering conflicting critical sections by scheduling priority, as in the MPCP, does not always yield the best results in terms of schedulability, and proposed using FIFO queues or semaphore-specific locking priorities instead. They further showed that assigning per-semaphore locking priorities that maximize schedulability

---

\(^1\) Under partitioned scheduling, each task is statically assigned to a processor, and each processor is scheduled individually using a uniprocessor policy.

\(^2\) The name “multiprocessor priority ceiling protocol” originally referred to the DPCP in [45], but was later repurposed to refer to the MPCP in [43]. We follow the terminology from [35,43,44], wherein the MPCP denotes the shared-memory variant.

\(^3\) Under global scheduling, all processors serve a shared ready queue and tasks migrate among all processors.
is an NP-complete problem [38]. More recently, Hsiu et al. [31] studied three problems related to finding task and resource assignments that minimize system-wide resource usage (i.e., the number of processors hosting real-time tasks, the number of processors hosting shared resources, and the total number of processors) assuming a distributed, priority-queue-based semaphore protocol similar to the DPCP. Unsurprisingly, they found the exact optimization problems to be intractable (NP-hard). In contrast to Hsiu et al.'s work [31], the notion of optimality studied herein focuses on the locking protocol itself (and not system-wide allocation properties), which makes it possible to find simple, asymptotically optimal solutions, as we show in Section 5.

Most closely related to this paper is [16], which was the first work to consider blocking optimality in (shared-memory) multiprocessor real-time systems, and from which we adopt the analytical framework and key definitions (as reviewed in detail in Section 2.3). In short, it was shown that even in the shared-memory case alone, there exist not only one, but two lower bounds on maximum blocking [16]. This is because there exist two sets of analysis assumptions, termed suspension-aware and suspension-oblivious schedulability analysis, that yield different lower bounds due to differences in how semaphore-related suspensions are accounted for during schedulability analysis. More precisely, in a system with \( m \) processors and \( n \) tasks, a lower bound of \( \Omega(n) \) was established in the suspension-aware case, whereas the suspension-oblivious case yields a lower bound of \( \Omega(m) \). In other words, it was shown that there exist pathological scenarios in which some tasks incur blocking that is (at least) linear in the number of processors (under suspension-oblivious) or linear in the number of tasks (under suspension-aware analysis), regardless of the employed locking protocol. These bounds have further been shown to be asymptotically tight with the construction of practical shared-memory semaphore protocols that ensure bounds on maximum blocking that are within a small constant factor of the established lower bounds [11, 13, 16, 18, 25, 49, 50].

To the best of our knowledge, no equivalent results are known for the case of distributed multiprocessor real-time semaphore protocols.4

1.3 Contributions

We study the question of optimal blocking in distributed multiprocessor real-time semaphore protocols and show that there exist two distinct task allocation strategies, which we call co-hosted and disjoint task allocation, that lead to different lower bounds on blocking. In the disjoint case, synchronization processors are dedicated exclusively to executing critical sections and may not host real-time tasks, whereas in the co-hosted case tasks also execute on synchronization processors. Notably, in a co-hosted scenario, we observe two surprising results:

1. in terms of the lower bound, there is no difference between suspension-aware and suspension-oblivious analysis, in contrast to the shared-memory case; and

2. blocking can be asymptotically worse than in an equivalent shared-memory system by a factor of \( \Phi \), where \( \Phi \) denotes the ratio of the maximum response time and the minimum period (formalized in Section 2)—we establish \( \Omega(\Phi \cdot n) \) as a lower bound on maximum blocking under both suspension-oblivious and suspension-aware analysis (Theorem 8).

We further show that any “reasonable” distributed locking protocol that does not artificially delay requests (formalized as “weakly work-conserving” in Section 2) causes at most \( \Theta(\Phi \cdot n) \) blocking (Theorem 10); any “reasonable” protocol is hence asymptotically optimal in the co-hosted case.

4 The material presented herein was previously made available online in preliminary form as an unpublished manuscript [12]. Based on [12], an in-kernel implementation and a fine-grained linear-programming-based blocking analysis of the protocol presented in Section 5.1 was previously discussed and evaluated in [14]. Whereas [14] focuses on accurate (non-asymptotic) analysis and practical concerns, the material presented in Sections 3–5 pertains exclusively to questions of optimality and has previously not been published.
In contrast to the co-hosted case, we show that, under disjoint task allocation, distributed locking protocols exist that ensure blocking bounds analogous to the shared-memory case: we establish lower bounds of $\Omega(n)$ and $\Omega(m)$ under suspension-aware and suspension-oblivious analysis, respectively, and show these bounds to be asymptotically tight by constructing two new distributed real-time semaphore protocols that ensure $O(n)$ and $O(m)$ maximum blocking under suspension-aware and suspension-oblivious analysis, respectively (Theorems 12 and 14).

The remainder of the paper is organized as follows. Section 2 provides essential definitions and a detailed review of the needed background. Section 3 establishes a lower bound on blocking with the construction of a task set that exhibits pathological blocking under co-hosted task allocation, and argues that prior constructions apply in the case of disjoint task allocation. Section 4 considers the co-hosted case and shows that any “reasonable” distributed locking protocol without artificial delays ensures maximum blocking within at most a constant factor of the established lower bound. Section 5 pertains to the case of disjoint task allocation and introduces two new protocols that establish the asymptotic tightness of the lower bounds under suspension-oblivious and suspension-aware analysis. Finally, Section 6 concludes with a discussion of the impact of communication delays.

2 Background and Definitions

In this section, we establish required definitions and review key prior results. In short, the results presented in this paper apply to sets of sporadic real-time tasks with arbitrary deadlines that are provisioned on a multiprocessor platform comprised of non-uniform processor clusters. Shared resources are accessible only from select synchronization processors and may be accessed from other processors using remote procedure calls (RPCs). These assumptions are formalized as follows; a summary of our notation is subsequently given at the end of the section in Table 1.

2.1 System Model

We consider the problem of scheduling a set of $n$ sporadic real-time tasks $\tau = \{T_1, \ldots, T_n\}$ on a set of $m$ processors. A sporadic task $T_i$ is characterized by its minimum inter-arrival separation (or period) $p_i$, its per-job worst-case execution time $e_i$, and its relative deadline $d_i$, where $e_i \leq \min\{d_i, p_i\}$. Each task releases a potentially infinite sequence of jobs, where two consecutive jobs of a task $T_i$ are released at least $p_i$ time units apart.

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 $t_a + d_i$. Both tasks and jobs are sequential: each job can be scheduled on at most one processor at a time, and a newly released job cannot be processed until the task’s previous job has been completed.

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 $\tau$ is schedulable if each $T_i \in \tau$ is schedulable. We define $\Phi$ to be the ratio of the maximum response time and the minimum period; formally $\Phi \triangleq \max\{r_i\}/\min\{p_i\}$.

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 $j$th 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 $K = m$ and $m_j = 1$ for each $C_j$. However, in general, clusters do not necessarily have a uniform size. We preclude the special case of $K = 1$ and $m_1 = m$ because distributed locking protocols are relevant only if there are at least two clusters (i.e., the case of $K = 1$ and $m_1 = m$ corresponds to a globally scheduled shared-memory platform, which is already covered by prior work [11,15,16,18]).
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 Section 3.

Each task \( T_i \) is statically assigned to one of the \( K \) clusters; we let \( C(T_i) \) denote \( T_i \)'s assigned cluster. Each cluster is scheduled independently according to a work-conserving job-level fixed-priority (JLFP) scheduling policy [21]. Two common JLFP policies are fixed-priority (FP) scheduling, where each task is assigned a fixed priority and jobs are prioritized in order of decreasing task priority, and earliest-deadline first (EDF) scheduling, where jobs are prioritized in order of decreasing absolute deadlines (with ties broken arbitrarily).

In general, 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 model includes the partitioned scheduling of shared-memory systems (each processor forms a singleton cluster). Each cluster may use a different JLFP policy. Our results apply to any JLFP policy.

Next, we discuss how resources may be shared in the assumed system architecture.

### 2.2 Distributed Real-Time Semaphore Protocols

In many real-time systems, tasks may have to share serially reusable resources (e.g., co-processors, I/O ports, shared data structures, etc.). This paper is concerned with systems in which mutually exclusive access to such resources is governed by a distributed (binary) semaphore protocol. In a distributed semaphore protocol, each resource can be accessed only from a (set of) designated processor(s); critical sections must hence be executed remotely if tasks use resources that are not local to their assigned processor.

We next formalize the assumed resource model and review a distributed semaphore protocol.

#### 2.2.1 Resource Model

The tasks in \( \tau \) are assumed to share \( n_r \) resources (besides the processors). Each shared resource \( \ell_q \) (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 \( \ell_q \) is local. Cluster \( C(\ell_q) \) is also called the synchronization cluster for \( \ell_q \).

To allow tasks to use non-local resources, access to each shared resource is mediated by one or more resource agents. To use a shared resource \( \ell_q \), a job \( J_i \) invokes an agent on cluster \( C(\ell_q) \) to carry out the request on \( J_i \)'s behalf using a synchronous RPC. After issuing an RPC, \( J_i \) suspends until notified by the invoked agent that the request has been carried out. A locking protocol such as the DPCP (reviewed in Section 2.2.2) determines how concurrent requests are serialized.

We let \( N_{i,q} \) denote the maximum number of times that any \( J_i \) uses \( \ell_q \), and let \( L_{i,q} \) denote the corresponding per-request maximum critical section length, that is, the maximum time that the agent handling \( J_i \)'s RPC requires exclusive access to \( \ell_q \) as part of carrying out any single operation invoked by \( J_i \). For notational convenience, we require \( L_{i,q} = 0 \) if \( N_{i,q} = 0 \) and define \( L_{i,q}^{\max} \triangleq \max\{L_{i,q} \mid 1 \leq q \leq n_r \land T_i \in \tau\} \).

Jobs invoke at most one agent at any time, and agents do not invoke other agents as part of handling a resource request (i.e., resource requests are not nested). An agent is active while it is processing requests, and inactive otherwise. While active, an agent is either ready (and can be
scheduled) or suspended (and is not available for execution). Active agents are typically ready, but may suspend temporarily when serving a request that involves synchronous I/O operations.

Following Rajkumar et al. [44,45], we assume that jobs can invoke agents without significant delay. That is, we assume that the overhead of cluster-to-cluster communication is negligible, in the sense that any practical system overheads can be incorporated into task parameters using standard overhead accounting techniques (e.g., see [11, Ch. 7]). If a distributed locking protocol is implemented on top of a platform with dedicated point-to-point links, or if the maximum communication delay across a shared network can be bounded by a constant (e.g., when communicating over a time-triggered network [34]), this assumption is appropriate, as any constant invocation cost can be accounted for using standard overhead accounting techniques. Further, such communication delays do not affect the blocking analysis per se (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 Section 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 readily handled using shared-memory protocols (or uniprocessor protocols) and are not the subject of this paper. We hence assume that each resource \( \ell_q \) is accessed by tasks from at least two different clusters.

Given our resource model, a locking protocol is required to determine how agents are prioritized, how conflicting requests are ordered, and when jobs may invoke agents. We next review the classic protocol for this purpose, namely the DPCP.

### 2.2.2 The Distributed Priority Ceiling Protocol

As the first (distributed) real-time semaphore protocol for multiprocessors, the DPCP [44,45] can be considered to be the prototypical distributed semaphore protocol for partitioned fixed-priority (P-FP) scheduling, a special case of the clustered JLFP scheduling assumed in this paper. We briefly review the DPCP as a concrete example of the considered class of protocols.

The DPCP fundamentally requires \( m_j = 1 \) for each cluster (or, rather, partition) \( C_j \). Each resource \( \ell_q \) is statically assigned to a specific processor and may not be directly used on other processors. Rather, tasks residing on other processors must indirectly access the resource by issuing RPCs to resource agents. To this end, the DPCP provides one resource agent \( A_{q,i} \) for each resource \( \ell_q \) and each task \( T_i \). To ensure a timely completion of critical sections, resource agents are subject to priority boosting, which means that they have priorities higher than any regular task (and thus cannot be preempted by regular jobs). Nonetheless, under the DPCP, 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 PCP [44,48]. 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 [44,48] is applied to prevent unbounded “priority inversion” (Section 2.3).

From an optimality point of view, not all of the details of the DPCP are relevant. Therefore, we abstract from the specifics of the DPCP in our analysis to consider a larger class of “DPCP-like” protocols, as defined next.
2.2.3 Simplified Protocol Assumptions

Specifically, in this paper, we focus on the class of distributed real-time locking protocols that ensure progress by means of two properties adopted from the DPCP [44,48].

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 $R$ for a resource $\ell_q$ is unsatisfied at time $t$ (i.e., $R$ has been issued but is not yet being processed) only if some resource (but not necessarily $\ell_q$) 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. Notably, the DPCP is only weakly work-conserving (and not work-conserving w.r.t. each resource) since requests for available resources may remain temporarily unsatisfied due to ceiling blocking [44,48].

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

Another simplification pertains to the use of 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 on 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.

Finally, we assume that there is only a single local agent for each resource. As seen in the DPCP [44,45], 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 agent activity corresponding to a resource $\ell_q$.

A key assumption in our system model is that both tasks and resources are statically assigned to clusters, which gives rise to two allocation scenarios, as we discuss next.

2.2.4 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 blocking bounds on maximum blocking in two separate scenarios, which we refer to as “co-hosted” and “disjoint” task allocation, respectively. Under co-hosted task allocation, 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, under disjoint task allocation, clusters may host either agents or tasks, but not both. The significance of these two allocation strategies is that they give rise to two distinct lower bounds on worst-case blocking, as will become apparent in Section 3.

Next, we give a precise definition of what actually constitutes “blocking.”

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 [44,48], 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), the term priority inversion blocking (pi-blocking) denotes any resource-sharing-related delay that affects worst-case response times [16]. 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 [16], 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 [46]), 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 pi-blocking to 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 does not increase the response time of any task (under preemptive JLF scheduling), but is also likely pessimistic as the processor load is lower in practice than assumed during analysis.

As 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 $\tau$ is schedulable under EDF on a uniprocessor if and only if $\sum_{\tau \in \mathcal{H}} \frac{r_{\tau}}{b_{\tau}} \leq 1$ [37]. 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 after inflating all execution costs $e_i$ by the maximum pi-blocking bounds $b_i$ [11,16,18]. That is, in the presence of locking-related self-suspensions, a set of resource-sharing, implicit-deadline sporadic tasks $\tau$ is schedulable under EDF on a uniprocessor if $\sum_{\tau \in \mathcal{H}} \frac{r_{\tau} + b_{\tau}}{b_{\tau}} \leq 1$.

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 [11,16,18], and because many published multiprocessor schedulability tests (e.g., [3–5,7–9,30]) do not account for self-suspensions explicitly.

In contrast, suspension-aware (s-aware) schedulability analysis explicitly accounts for all effects of pi-blocking. For instance, response-time analysis (RTA) for (uniprocessor) FP scheduling [1,35] is a good example of effective s-aware schedulability analysis, and can be applied to partitioned scheduling as follows. Let $b_i^c$ 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 $b_i = b_i^c + b_i^l$. 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 positive solution to the recursion [1,35]

\[
    r_i = e_i + b_i^c + b_i^l + \sum_{T_h \in hp(T_i)} \left[ \frac{r_i + b_h^c}{p_h} \right] \cdot e_h, \tag{1}
\]

where $hp(T_i)$ denotes the set of tasks assigned to processor $C(T_i)$ with higher priorities than $T_i$. Equation (1) is an s-aware schedulability test because $b_i = b_i^c + b_i^l$ is explicitly accounted for.
This difference—explicit vs. implicit suspension accounting—has a profound impact on the exact nature of pi-blocking, as we review next.

### 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-priority job.\(^5\) Prior work \([11, 16, 18]\) has shown that, since s-oblivious schedulability analysis over-approximates a task’s processor demand, the definition of “priority inversion” depends on the type of analysis.

▶ **Definition 1.** Under *s-oblivious* schedulability analysis, a job \(J_i\) of a task \(T_i\) assigned to cluster \(C_j = C(T_i)\) incurs *s-oblivious pi-blocking* at time \(t\) if \(J_i\) is pending but not scheduled and fewer than \(m_j\) higher-priority jobs of tasks assigned to \(C_j\) are *pending* \([16]\).

▶ **Definition 2.** Under *s-aware* schedulability analysis, a job \(J_i\) of a task \(T_i\) assigned to cluster \(C_j = C(T_i)\) incurs *s-aware pi-blocking* at time \(t\) if \(J_i\) is pending but not scheduled and fewer than \(m_j\) higher-priority ready jobs of tasks assigned to \(C_j\) are *scheduled* \([16]\).

Note that there cannot be fewer pending higher-priority jobs than there are scheduled higher-priority jobs (i.e., a scheduled job is necessarily also pending). Hence, if a job \(J_i\) incurs s-oblivious pi-blocking at a time \(t\), then it incurs also s-aware pi-blocking at time \(t\). However, the converse does not hold: if \(J_i\) incurs s-aware pi-blocking time \(t\), then it may be the case that it does *not* incur s-oblivious pi-blocking at time \(t\). More precisely, \(J_i\) incurs s-aware pi-blocking, but not s-oblivious pi-blocking, at time \(t\) if there are at least \(m_j\) higher-priority jobs pending, but fewer than \(m_j\) of them are scheduled at time \(t\).

In other words, if Definition 1 is satisfied, then Definition 2 is satisfied as well. Therefore, an upper bound on s-aware pi-blocking (Definition 2) implies an upper bound on s-oblivious pi-blocking (Definition 1), as previously pointed out in \([16]\). Conversely, a lower bound on s-oblivious pi-blocking (Definition 1) also implies a lower bound on s-aware pi-blocking (Definition 2). We use this relationship in Section 3.

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 protocol \([11, 16]\).

### 2.3.3 PI-Blocking Complexity

As discussed in Section 1, blocking optimality is concerned with finding the smallest possible bound on worst-case blocking. To enable systematic study of this question, *maximum pi-blocking*, formally \(\max\{b_i \mid T_i \in \tau\}\), has been proposed as a metric of blocking complexity in prior work \([11, 16, 18]\).

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 \(T_i\), \(\sum_{1 \leq q \leq n} N_{i,q}\) and each \(L_{i,q}\) to be constants and assume \(n \geq m\). All other parameters are considered variable (or dependent on \(m\) and \(n\)).

---

\(^5\) Regular interference due to the scheduling of higher-priority jobs is accounted for by any sound schedulability test. A priority inversion exists if *additional* delay is incurred.
We start by establishing a lower bound on maximum s-oblivious and s-aware pi-blocking in the case of co-hosted task allocation. To this end, we establish the existence of pathological task sets in which some task always incurs resource \( \ell \).

### Table 1 Summary of notation.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m )</td>
<td>total number of processors</td>
</tr>
<tr>
<td>( K )</td>
<td>number of clusters, ( 2 \leq K \leq m )</td>
</tr>
<tr>
<td>( C_j )</td>
<td>the ( j )th cluster, ( 1 \leq j \leq K )</td>
</tr>
<tr>
<td>( m_j )</td>
<td>number of processors in ( C_j )</td>
</tr>
<tr>
<td>( n )</td>
<td>total number of tasks</td>
</tr>
<tr>
<td>( T_i )</td>
<td>the ( i )th sporadic task, ( 1 \leq i \leq n )</td>
</tr>
<tr>
<td>( J_i )</td>
<td>a job of ( T_i )</td>
</tr>
<tr>
<td>( e_i )</td>
<td>( T_i )'s WCET</td>
</tr>
<tr>
<td>( p_i )</td>
<td>( T_i )'s period</td>
</tr>
<tr>
<td>( d_i )</td>
<td>( T_i )'s relative deadline</td>
</tr>
<tr>
<td>( r_i )</td>
<td>( T_i )'s max. response time</td>
</tr>
<tr>
<td>( C(T_i) )</td>
<td>( T_i )'s assigned cluster</td>
</tr>
<tr>
<td>( n_r )</td>
<td>number of shared resources</td>
</tr>
<tr>
<td>( \ell_q )</td>
<td>the ( q )th shared resource, ( 1 \leq q \leq n_r )</td>
</tr>
<tr>
<td>( A_q )</td>
<td>the agent handling requests for ( \ell_q )</td>
</tr>
<tr>
<td>( C(\ell_q) )</td>
<td>cluster to which ( \ell_q ) is local</td>
</tr>
<tr>
<td>( N_{i,q} )</td>
<td>max. number of requests of any ( J_i ) for ( \ell_q )</td>
</tr>
<tr>
<td>( L_{i,q} )</td>
<td>max. critical section length of ( T_i ) w.r.t. ( \ell_q )</td>
</tr>
<tr>
<td>( L_{i,q}^{\text{max}} )</td>
<td>max. ( L_{i,q} ) for any ( T_i ) and any ( \ell_q )</td>
</tr>
<tr>
<td>( b_i )</td>
<td>max. pi-blocking incurred by any ( J_i )</td>
</tr>
<tr>
<td>( \Phi )</td>
<td>ratio of the longest max. response time of any ( T_i ) and the shortest period of any ( T_i )</td>
</tr>
</tbody>
</table>

Under these assumptions, it was shown \([11,16,18]\) 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 \( m \) (and independent of the number of tasks \( n \)) under s-oblivious analysis, but linear in \( n \) (and independent of \( m \)) under s-aware analysis \([11,16,18]\). 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. In other words, in the case of shared-memory semaphore protocols, the real-time mutual exclusion problem can be solved such that \( \max \{ b_i \mid T_i \in \tau \} = \Theta(m) \) under s-oblivious schedulability analysis, and such that \( \max \{ b_i \mid T_i \in \tau \} = \Theta(n) \) under s-aware schedulability analysis \([11,16,18]\).

We can now precisely state the contribution of this paper: in the following sections, we establish upper and lower bounds on \( \max \{ b_i \mid T_i \in \tau \} \) under s-oblivious and s-aware schedulability analysis for distributed (i.e., DPCP-like) real-time locking protocols, thereby complementing the earlier results on shared-memory (i.e., MPCP-like) real-time locking protocols \([11,16,18]\). For ease of reference, the notation used in this paper is summarized in Table 1.

## 3 Lower Bounds on Maximum PI-Blocking

We start by establishing a lower bound on maximum s-oblivious and s-aware pi-blocking in the case of co-hosted task allocation. To establish a general lower bound, it is sufficient to construct an example task set that demonstrates that the claimed amount of pi-blocking (either s-aware or s-oblivious) is always possible under any locking protocol compliant with Assumptions A1 and A2. To this end, we establish the existence of pathological task sets in which some task always incurs \( \Omega(\Phi \cdot n) \) pi-blocking due to priority boosting (Assumption A1), regardless of whether s-oblivious or s-aware schedulability analysis is used. This family of task sets is defined as follows.

### Definition 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 2, that share one resource \( \ell_1 \) local to cluster \( C_1 \) (i.e., \( C(\ell_1) = C_1 \)).
Table 2: 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 $C_1$. Recall that $n \geq m \geq 2 \cdot m_1$.

<table>
<thead>
<tr>
<th>$e_i$</th>
<th>$p_i$</th>
<th>$d_i$</th>
<th>$N_{i,1}$</th>
<th>$L_{i,q}$</th>
<th>$C(T_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{R \cdot n}{2}$</td>
<td>$R \cdot n$</td>
<td>$R \cdot n$</td>
<td>0</td>
<td>0</td>
<td>$C_1$ for $i \in {1, \ldots, m_1}$</td>
</tr>
<tr>
<td>$a$</td>
<td>$n$</td>
<td>$n + 1$</td>
<td>1</td>
<td>$b$</td>
<td>$C_{(2+i \mod (K-1))}$ for $i \in {m_1 + 1, \ldots, 2 \cdot m_1}$</td>
</tr>
<tr>
<td>$a$</td>
<td>$n$</td>
<td>$n + 1$</td>
<td>1</td>
<td>$a$</td>
<td>$C_{(2+i \mod (K-1))}$ for $i &gt; 2 \cdot m_1$ (if any)</td>
</tr>
</tbody>
</table>

Figure 1: Example schedule of the task set $\tau^{seq}(n, m_1, R)$ as specified in Table 2 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 $\ell_1$, 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.

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 2 that the maximum critical section lengths (w.r.t. $\ell_1$) 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 tasks are a special case of sporadic tasks and thus sufficient to establish a lower bound.

For simplicity and without loss of generality, we further assume that each job of tasks $T_{m_1+1}, \ldots, T_n$ immediately accesses resource $\ell_1$ 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. Figure 1 depicts an example schedule for $K = 2$, $m_1 = 2$, $m_2 = 2$, $n = 5$, and $R = 3$.

We begin by observing that the agent servicing requests for $\ell_1$, denoted $A_1$ in the following, continuously occupies one of the processors in cluster $C_1$.

Lemma 4. If $m_1 > 1$, then only $m_1 - 1$ processors of cluster $C_1$ service jobs of tasks $T_1, \ldots, T_{m_1}$.
Proof. By construction, the agent $A_1$ servicing requests for resource $\ell_1$ 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 $\ell_1$ 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 job at time $t_a$. Upon being scheduled, each such job immediately accesses resource $\ell_1$ and suspends until its request is serviced. As a result, regardless of the JLF 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

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

time units during the interval $[t_a, t_a + n)$, regardless of how the employed locking protocol serializes requests for $\ell_1$. 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 $T_1, \ldots, T_{m_1}$ under any JLF policy and any distributed locking protocol satisfying assumptions A1 and A2.

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

Lemma 5. 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) JLF 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 $C_1$. Recall that we assume that priorities are unique (i.e., any ties in priorities are subject to arbitrary but consistent tie-breaking). Let $J_i$ denote this lowest-priority job. By Lemma 4, there are only $m_1 - 1$ processors available to service jobs. Thus $J_i$ 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 $e_i = \frac{R \cdot n}{C}$, in the worst case, job $J_i$ is not scheduled until time $t_a + \frac{R \cdot n}{C}$, and then requires another $e_i = \frac{R \cdot n}{C}$ 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 6. 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 2 that the maximum critical section lengths of tasks $T_{m_1+1}, \ldots, T_n$ are halved if $m_1 = 1$. Analogously to Lemma 4, it can thus be shown that, in the worst case, the single processor in $C_1$ is available to $T_1$ for only $\frac{R}{2}$ 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 $\frac{R}{2}$ time units every $n$ time units), $J_1$ will accumulate $e_1$ time units of processor service only by time $t_a + 2e_1 = t_a + R \cdot n$. ▶
Since there are $m_1$ processors and $m_1$ pending jobs in cluster $C_1$, all pending jobs should be immediately scheduled under any work-conserving scheduling policy. However, since the priority-boosted agent occupies one of the processors, this is not the case, which implies that one job incurs s-oblivious pi-blocking (under any work-conserving JLFP policy).

**Lemma 7.** Under 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. Hence any delay of a pending job constitutes s-oblivious pi-blocking (recall Definition 1): \( b_i = r_i - c_i \) for each \( T_i \in \{T_1, \ldots, T_{m_1}\} \), regardless of the employed JLFP scheduling policy. Since \( c_i = \frac{R \cdot n}{2} \) 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 5 and 6, \( \max \{ r_i \mid 1 \leq i \leq m_1 \} = R \cdot n \), 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 under co-hosted task allocation.

**Theorem 8.** Under JLFP scheduling, using either s-aware or s-oblivious schedulability analysis, there exists a task set such that, under co-hosted task allocation, \( \max \{ b_i \} = \Omega(\Phi \cdot n) \) under any weakly work-conserving distributed multiprocessor real-time semaphore protocol that employs priority-boosted agents (i.e., under protocols matching Assumptions A1 and A2).

**Proof.** By Lemma 7, there exists a task set $\tau^{seq}(n, m_1, R)$ such that, under s-oblivious schedulability analysis, any JLFP policy, and any distributed multiprocessor semaphore protocol satisfying Assumptions A1 and A2, \( \max \{ b_i \mid T_i \in \tau^{seq}(n, m_1, R) \} \geq \frac{R \cdot n}{2} \) for any \( R \in \mathbb{N} \). Recall from Section 2.1 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.

Recall from Section 2.3 that s-oblivious pi-blocking implies s-aware pi-blocking (i.e., Definition 2 holds if Definition 1 is satisfied). The established lower bound on s-oblivious pi-blocking therefore also applies to s-aware pi-blocking [16], and thus \( \max \{ b_i \} = \Omega(\Phi \cdot n) \) under either s-aware or s-oblivious schedulability analysis.

Compared to a shared-memory system, where the shared-memory mutual exclusion problem can be solved with $\Theta(n)$ maximum s-aware pi-blocking in the general case [11, 16], Theorem 8 shows that maximum pi-blocking under distributed locking protocols is asymptotically worse by a factor of $\Phi$. Maximum s-oblivious pi-blocking is also asymptotically worse—the equivalent shared-memory mutual exclusion problem can be solved with $\Theta(m)$ maximum s-oblivious pi-blocking [11, 16, 18] (recall that we assume $n \geq m$). Note that, $\Phi$, the ratio of the maximum response time and the minimum period, can in general 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 in a distributed environment.

The observed discrepancy, however, is entirely due to the effects of preemptions caused by priority-boosted agents. While it is not possible to avoid priority boosting entirely (otherwise excessive pi-blocking could result when agents are preempted by jobs with large execution costs), such troublesome preemptions can be easily ruled out by disallowing the co-hosting of agents and tasks in the same cluster. And in fact, when using such a disjoint task allocation approach, the asymptotic lower bounds on maximum pi-blocking under distributed locking protocols are identical to those previously established for shared-memory semaphore protocols. The matching lower bounds can be trivially established with the setup previously used in [16]; we omit the details here and summarize the correspondence with the following theorem.
Theorem 9. There exist task sets such that, under JLFP scheduling, disjoint task allocation, and any distributed real-time semaphore protocol satisfying 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.

Having established lower bounds on s-oblivious and s-aware pi-blocking under both co-hosted and disjoint task allocation, we next explore the question of asymptotic optimality—how to construct protocols that ensure upper bounds on maximum pi-blocking that are within a constant factor of the established lower bounds? We begin with the co-hosted case in Section 4, and consider the disjoint case in Section 5 thereafter.

As a final remark, we note that the task set \(\tau^{seq}(n, m_1, R)\) as given in Table 2 contains tasks with relative deadlines larger than periods (i.e., \(d_i > p_i \) for \(i > m_1\)). This is purely a matter of convenience; asymptotically equivalent bounds can be derived with implicit-deadline tasks.

4 Asymptotic Optimality under Co-Hosted Task Allocation

Theorem 8 shows that there exist pathological scenarios in which the choice of real-time locking protocol is seemingly irrelevant: regardless of the specifics of the employed locking protocol, worst-case pi-blocking is asymptotically worse than in a comparable shared-memory system simply because resources are inaccessible from some processors. Curiously, from an asymptotic point of view, protocol-specific rules are indeed immaterial: any distributed real-time locking protocol that does not starve requests is asymptotically optimal in the case of co-hosted task allocation.

Theorem 10. 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. Recall from Definition 2 that a pending job \(J_b\) incurs s-aware pi-blocking if \(J_b\) is not scheduled and not all processors in its assigned cluster are occupied by higher-priority jobs. This happens either when (i) \(J_b\) is suspended while waiting for a resource request to be completed, or when (ii) \(J_b\) is preempted by a priority-boosted agent that executes on behalf of 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 a trivial upper bound on the maximum cumulative agent activity, and hence also on the maximum total duration of pi-blocking incurred by \(J_b\).

To this end, consider for any task \(T_x\) the maximum number of jobs of \(T_x\) that execute while \(J_b\) is pending, which is bounded by \(\left\lfloor \frac{r_x + r_b}{p_x} \right\rfloor\). Since there are \(n\) tasks in total, this implies that at most \(\sum_{x=1}^{n} \left\lfloor \frac{r_x + r_b}{p_x} \right\rfloor = \sum_{x=1}^{n} \left( \frac{r_x}{p_x} + \frac{r_b}{p_x} \right) \leq \sum_{x=1}^{n} \left( \frac{\max_i(r_i)}{\min_i(p_i)} + \frac{\max_i(r_i)}{\min_i(p_i)} \right) = n \cdot \left\lceil 2\Phi \right\rceil = O(\Phi \cdot n)\) jobs (in total across all tasks) are executed while \(J_b\) is pending. Since \(\sum_i N_{i,q} \cdot L_{i,q} = O(1)\) for each \(T_i\) (i.e., since each job issues at most a constant number of requests), it follows that \(\max_i(b_i) = O(n \cdot \Phi)\), regardless of any protocol-specific rules.

Recall from Section 2.3 that s-oblivious pi-blocking implies s-aware pi-blocking (i.e., if Definition 1 is satisfied, then Definition 2 holds, too). Hence, an upper bound on s-aware pi-blocking

\[6\] See e.g. [11, Ch. 4] for a formal proof of this well-known bound.
explicitly also upper-bounds s-oblivious pi-blocking, and thus \( \max_i \{ b_i \} = O(n \cdot \Phi) \) under either s-oblivious or s-aware schedulability analysis.

As a corollary, Theorem 10 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 as the queue order is crucial in the shared-memory case [16].

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 tasks and critical sections are necessarily co-hosted, i.e., executed on the same set of processors)—it is indeed possible to ensure maximum s-aware pi-blocking that is asymptotically no worse than under a shared-memory locking protocol. We establish this fact next by introducing two new protocols that realize \( O(n) \) and \( O(m) \) maximum pi-blocking under s-aware and s-oblivious schedulability analysis, respectively, in the case of disjoint task allocation. As one might expect, the choice of queue order is significant in this case.

## 5 Asymptotic Optimality under Disjoint Task Allocation

Prior work [11, 15, 16, 18] 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 \textit{FIFO Multiprocessor Locking Protocol} (FMLP+) for s-aware analysis [11, 15] and the family of \textit{O-MLP Locking Protocols} (the OMLP family) for s-oblivious analysis [11, 16, 18], 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 Asymptotically Optimal Maximum S-Aware PI-Blocking

Inspired by the FMLP+ [11], the \textit{Distributed FIFO Locking Protocol} (DFLP) relies on simple FIFO queues to avoid starvation. Notably, the DFLP ensures \( O(n) \) maximum s-aware pi-blocking under disjoint task allocation 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 any \( j \neq h \)). We first describe the structure and rules of the DFLP, and then establish its asymptotic optimality.

#### 5.1.1 Rules

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

1. When \( J_i \) issues a request \( R \) for resource \( \ell_q \), \( J_i \) suspends and \( R \) is appended to \( FQ_{\ell_q} \). \( J_i \)’s request is processed by agent \( A_q \) when \( R \) becomes the head of \( FQ_{\ell_q} \).
2. When \( R \) is complete, it is removed from \( FQ_{\ell_q} \) and \( J_i \) is resumed.
3. Active agents are scheduled preemptively in the order in which their current requests were issued (i.e., an agent processing an earlier-issued request has higher priority than one serving a later-issued request). Any ties 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.
5.1.2 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 10 hence applies.

To show asymptotic optimality in the disjoint case, we first establish a per-request bound on the number of interfering requests that derives from FIFO-ordering both requests and agents.

▶ **Lemma 11.** Let $\mathcal{R}$ denote a request issued by a job $J_i$ for a resource $\ell_q$ and let $T_x$ denote a task other than $T_i$ (i.e., $i \neq x$). 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. Since $\mathcal{R}$ is not delayed by later-issued requests (and clearly not by earlier-completed requests), all requests that delay the completion of $\mathcal{R}$ are incomplete at the time that $\mathcal{R}$ is issued. Since tasks and jobs are sequential, and since jobs request at most one resource at a time, there exists at most one incomplete request per task at any time. ◀

An $O(n)$ bound on maximum s-aware 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 cannot preempt jobs in the disjoint setting.

▶ **Theorem 12.** Under the DFLP with disjoint task allocation, $\max \{b_i\} = O(n)$.

**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 11, each other task delays each of $J_i$’s $\sum_q N_{i,q}$ requests for at most the duration of one request, that is, per request, $J_i$ incurs no more than $n \cdot L_{\text{max}}$ s-aware pi-blocking. Since $J_i$ issues at most $\sum_q N_{i,q}$ requests, and since by assumption $\sum_q N_{i,q} = O(1)$ and $L_{\text{max}} = O(1)$, we have $b_i \leq n \cdot L_{\text{max}} \cdot \sum_q N_{i,q} = O(n)$. ◀

The DFLP is thus asymptotically optimal with regard to maximum s-aware pi-blocking, under both co-hosted (Theorem 10) and disjoint task allocation (Theorem 12). In contrast, the DPCP does not generally guarantee $O(n)$ s-aware pi-blocking in the disjoint case since it orders conflicting requests by task priority and is thus prone to starvation issues (this can be shown similarly to the lower bound on priority queues established in [11,16]).

This concludes the case of s-aware analysis. Next, we consider the s-oblivious case.

5.2 Asymptotically 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 distributed systems.

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: if an agent is faced with $n$ concurrent requests, it is not possible to ensure $O(m)$ maximum s-oblivious pi-blocking regardless of the order in which requests are processed. 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 on only 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 [16]. 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 [18, 25, 50], including a variant of the OMLP [18]; the contention tokens can thus be readily managed within each cluster using any of the available \( k \)-exclusion protocols [18, 25, 50]. These considerations lead to the following protocol definition.

### 5.2.1 Rules

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

Jobs may access shared resources according to the following rules. In the following, let \( J_i \) denote a job that must access resource \( \ell_q \).

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.

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

3. When \( R \) is complete, it is removed from \( FQ_q \). \( J_i \) is resumed and immediately relinquishes its contention token.

4. Active, ready agents are scheduled preemptively in order of non-decreasing request enqueueing times (i.e., while processing \( R \), agent \( A_q \)'s priority is the point in time at which \( R \) 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).

As shown next, the contention tokens in combination with FIFO-ordering requests and agents yield an asymptotically optimal maximum s-oblivious pi-blocking bound.

### 5.2.2 Blocking Complexity

As with the DFLP, the co-hosted case is trivial since Theorem 10 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 \( R \) is preceded by other requests in \( FQ_q \) or if \( A_q \) is preempted while processing \( R \)).

▶ **Lemma 13.** A job \( J_i \) holds a contention token for at most \( m \cdot L^\text{max} \) time units per request.

**Proof.** By Rules 1 and 3, a job \( J_i \) holds a contention token while it waits for its request \( R \) to be completed. Analogously to Lemma 11, since \( FQ_q \) is FIFO-ordered and since agents are scheduled in FIFO order w.r.t. the time that requests are enqueued (Rule 4), the completion of \( R \) can only
be delayed due to the execution of requests that were incomplete at the time that \( R \) 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 \( R \) is enqueued in \( FQ_q \). Hence, \( R \) is completed and \( J_i \) relinquishes its contention token after at most \( m \cdot L^{\text{max}} \) time units.

By leveraging a \( k \)-exclusion protocol that is asymptotically optimal under s-oblivious analysis (Rule 1), Lemma 13 immediately yields an \( O(m) \) bound on maximum s-oblivious pi-blocking.

\[ \textbf{Theorem 14.} \text{ Under the D-OMLP with disjoint task allocation, } \max\{b_i\} = O(m). \]

\textbf{Proof.} Let \( H \) denote the maximum token-hold time. By Lemma 13, the maximum token-hold time is \( H = m \cdot L^{\text{max}} = O(m) \). 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 \( O \left( \frac{m}{m_j} \right) \) critical section lengths per request \([18,25,50]\). Under the D-OMLP, there are \( k = m_j \) contention tokens in each cluster \( C_j \). Hence, in the disjoint setting, a task assigned to cluster \( C_j \) incurs \( O \left( \frac{m}{m_j} \cdot H \right) = O(H) = O(m) \) 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.

\section{Conclusion}

In this paper, we studied blocking optimality in distributed real-time locking protocols. We identified two different task and resource allocation strategies, namely co-hosted and disjoint task allocation, that give rise to different answers to this question. In the co-hosted case, under both s-aware and s-oblivious analysis, \( \Omega(\Phi \cdot n) \) maximum pi-blocking is unavoidable in the general case, whereas in the disjoint case, \( \Omega(n) \) maximum s-aware and \( \Omega(m) \) maximum s-oblivious pi-blocking are the fundamental lower bounds. The significance of these bounds is that the lower bound on maximum pi-blocking in the case of co-hosted task allocation is asymptotically worse than in an equivalent shared-memory scenario. In contrast, disjoint task allocation yields the same lower bounds already known from the analysis of shared-memory synchronization.

We further showed that the established lower bounds are asymptotically tight. In the co-hosted case, any distributed locking protocol satisfying Assumptions A1 and A2 is asymptotically optimal (Theorem 10). To prove asymptotic tightness in the disjoint case, we introduced two new distributed real-time semaphore protocols. Specifically, the DFLP is asymptotically optimal under s-aware analysis, and the D-OMLP is asymptotically optimal under s-oblivious analysis, both w.r.t. the maximum pi-blocking metric.

Pi-blocking is generally undesirable, and hence protocols that guarantee lower asymptotic pi-blocking bounds are intuitively preferable. Our results are the first formal characterization of the fundamental limits on pi-blocking in a distributed setting and serve to structure the design space of distributed real-time locking protocols. However, one should also note that a lower asymptotic pi-blocking bound does not necessarily imply better overall schedulability.

For one, while disjoint task allocation permits lower bounds on pi-blocking, it also requires dedicating some cluster(s) to agents, which, depending on constant factors such as the level of contention and critical section lengths, may decrease the overall utilization of the system. Whether disjoint task allocation is beneficial is thus a workload-specific question that must be answered individually for each task set.
Further, asymptotic optimality does not imply that an asymptotically optimal protocol is always preferable to a non-optimal one. Rather, blocking bounds of asymptotically similar locking protocols can still differ significantly 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 under co-hosted task allocation, 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 request frequencies. For the DFLP and the DPCP, we have recently developed such bounds [14]; the same techniques could also be applied to analyze the D-OMLP.

As noted in Section 2.2, 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 [11]). This is a reasonable assumption in platforms with point-to-point links, in systems with networks employing TDMA or time-triggered [34] arbitration policies, or if distributed semaphore protocols are implemented on top of a (large) shared-memory platform (e.g., see [14] for such a case). However, the assumption may be more problematic in systems that require explicit message routing across a shared, dynamically arbitrated network. Assuming there exists an upper bound $Δ_{i,q}$ on the message delay between a task $T_i$ and each agent $A_q$, such delays can be incorporated by simply increasing $T_i$’s self-suspension time by $2Δ_{i,q}$ for each agent invocation (under the D-OMLP, the maximum token-hold time is increased by $2Δ_{i,q}$ as well). If $Δ_{i,q}$ can be considered constant (i.e., if $Δ_{i,q} = O(1)$ from an asymptotic analysis point of view), then the asymptotic upper and lower bounds established in this paper remain unaffected. If, however, $Δ_{i,q}$ depends on $m$ or $n$, or on other non-constant factors, then additional analysis is required, which may be an interesting direction for future work.

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 [49]; however, it remains to be seen how their techniques can be extended to distributed real-time semaphore protocols.

References

Björn B. Brandenburg


