Fine-Grained Multiprocessor Real-Time Locking with Improved Blocking [*]

Bryan C. Ward and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

Existing multiprocessor real-time locking protocols that support nesting are subject to adverse blocking that can be avoided when additional resource-usage-pattern information is known. These sources of blocking stem from system overheads, varying critical section lengths, and a lack of support for replicated resources. In this paper, these issues are resolved in the context of the recently proposed real-time nested locking protocol (RNLP). The resulting protocols are the first to support fine-grained real-time lock nesting while allowing multiple resources to be locked in one atomic operation, both spin- and suspension-based waiting to be used together, and resources to be replicated. They also reduce "short-on-long" blocking, which is very detrimental if both very long and very short critical sections must be supported.

1 Introduction

In concurrent systems, it is sometimes necessary for a single task to perform operations on multiple shared resources concurrently. When lock-based mechanisms are used to realize resource sharing, such concurrent operations can be implemented by *nesting* lock requests. In this paper, we consider multiprocessor systems that employ lock nesting and that also have real-time constraints. In this case, a synchronization protocol must be used that, when coupled with a scheduling algorithm, ensures that all timing constraints can be met.

There currently exist two general techniques for supporting nested resource requests on multiprocessor real-time systems: coarse- and fine-grained locking. Under coarse-grained locking, resources that may be accessed in a nested fashion are grouped into a single lockable entity, and a single-resource locking protocol is used. This approach is also known as *group locking* [1]. In contrast, a fine-grained locking protocol allows such resources to be held concurrently by different tasks [12]. In recent work, we developed the first such protocol for multiprocessor real-time systems: the *real-time nested locking protocol* (*RNLP*) [12]. The RNLP is actually a "pluggable" protocol that has different variants for different schedulers and analysis assumptions. Most of these variants are asymptotically optimal with respect to blocking behavior.

[*]Work supported by NSF grants CNS 1016954, CNS 1115284, CNS 1218693, and CNS 1239135; and ARO grant W911NF-09-1-0535. The first author was supported by an NSF graduate research fellowship.

Fine-grained locking often allows for increased parallelism among resource-using tasks. If this parallelism can be captured analytically, then predicted worst-case blocking times decrease. However, even if more pessimistic blocking analysis is applied, the increased parallelism afforded by fine-grained lock nesting allows for improved response times in practice. Also, fine-grained lock nesting is more dynamic in that resources can be more easily added to or removed from a system. In contrast, under coarse-grained locking, resource groups must be statically created before execution.

After developing the RNLP, we attempted to apply it within two interesting use cases. In the first, we sought to manage usage of *graphics processing units* (*GPUs*) by employing fine-grained locking to arbitrate access to various interconnects and GPU functional units that are involved in GPU computations [9]. In the second, we sought to manage the shared cache of a multicore machine by treating cache lines as shared resources that tasks may acquire via a fine-grained locking protocol [14]. In both use cases, we found that, despite its asymptotic optimality, the RNLP can sometimes cause unnecessary or problematic blocking:

I1 If a task requires access to multiple resources that are *a priori* known, then acquiring each resource individually in a nested fashion can unnecessarily increase system-call overhead and hence blocking times for suspension-based locks. This is especially problematic if such overheads are long relative to critical section lengths.

I2 In the RNLP, requests may be blocked by other possibly non-conflicting requests for different resources. This can cause *short-on-long* blocking, i.e., short requests may be blocked by long requests. This is particularly problematic if critical section lengths are highly variant or if some critical sections are quite lengthy, as is true in our GPU use case.

I3 In applications in which resources (e.g., GPUs) are *replicated*, viewing each replica as a distinct resource may cause unnecessary blocking if a task merely requires access to *some* replica and not a specific one. The original RNLP does not support replicated resources.

While Issue I3 concerns new functionality, Issues I1 and I2 stem from the fact that the RNLP was designed with asymptotic optimality in mind: in designing it, system overheads were ignored, and critical section lengths were considered constants (thus, effectively ignored). In this paper, we explain how all three issues can be addressed to obtain new protocols with better blocking bounds. As discussed elsewhere [9, 14], several of these new protocols have been successfully applied in the GPU and shared-cache use cases mentioned above.

Prior work. Rajkumar developed the first multiprocessor real-time locking protocols, the *multiprocessor priority ceiling protocol* (*MPCP*) and the *distributed priority ceiling protocol* (*DPCP*) [11]. More recently, these results have been built upon to produce the *MPCP with virtual spinning* (*MPCP-VS*) [10] and the *parallel priority ceiling protocol* (*PPCP*) [6]. However, the prior work most applicable to the issues we address is the *flexible multiprocessor locking protocol* (FMLP) and related protocols [1, 2]. Under the FMLP, resources are categorized as *short* or *long* depending on access times; tasks wait on short resources by spinning and on long resources by suspending. The FMLP allows requests for short resources to be nested within requests for long resources, but not *vice versa*. This eliminates short-on-long blocking. Also, the FMLP uses group locks to support the nesting of requests that are either all short or all long.

Contributions. In this paper, we address the three issues raised above for job-level fixed priority (JLFP) systems, i.e., systems in which each job has a constant priority. We address Issue I1 by allowing a task to lock multiple resources with one lock request—we call such locks *dynamic group locks* (*DGLs*). DGLs are a hybrid of coarse- and fine-grained locking in that a task need not request an entire group of resources, but rather only the subset it requires. Also, tasks may issue nested DGL requests.

We address Issue I2 by enabling short requests to be satisfied more greedily. This can cause additional blocking on short requests by long requests (*long-on-short* blocking), but this is often an acceptable tradeoff at runtime. We allow waiting on short resources to be done by either spinning or suspending; thus, we allow both waiting mechanisms to be used in the same protocol. In the original RNLP, different waiting mechanisms are not used together.

Finally, we address Issue I3 by introducing support for replicated resources in the RNLP, which requires altering some of its queue structures to allow multiple tasks to hold replicas of the same resource concurrently. The resulting parallelism is reflected in the blocking analysis.

Organization. In Secs. 2–3, we present background material and review the RNLP. We then present our extensions of the RNLP in Secs. 4–6. In Sec. 7, we present an experimental evaluation, and in Sec. 8, we conclude.

2 Background and Definitions

We assume the sporadic task model in which there are n tasks $\tau = \{T_1, \ldots, T_n\}$ that execute on m processors. We denote the k^{th} job (invocation) of the i^{th} task as $J_{i,k}$ though we often omit the job index k if it is insignificant. Each task T_i is characterized by a *worst-case execution time* e_i, *minimum job separation* p_i, and *relative deadline* d_i. For simplicity, we assume implicit deadlines ($d_i = p_i$), and that every job must complete before its deadline (no tardiness). We say that a released job is *pending* until it finishes its execution.

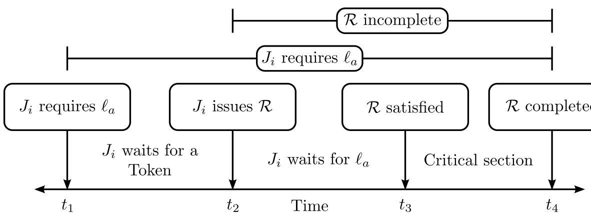


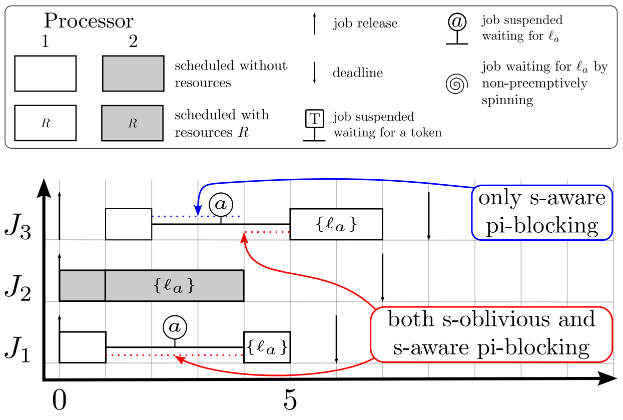
Figure 1: Illustration of request phases.

Resources. We consider a system that contains q shared resources $\mathcal{L} = \{\ell_1, \ldots, \ell_q\}$. We assume basic familiarity with terms related to resource sharing (e.g., critical section, outermost critical section, etc.). With respect to the RNLP (see Sec. 3), resource requests proceed through several phases, as depicted in Fig. 1. A job making an outermost request must first acquire a token, as described in Sec. 3. Once a token is acquired, resources may be requested in a nested fashion. Once such a request is *issued*, the requesting job blocks (if necessary) until the request is *satisfied*, and then continues to hold the requested resource until its critical section is *completed*. An issued but not completed request is called an *incomplete request*. A job that has an incomplete request and is waiting for a shared resource is said to have an *outstanding resource request*. Waiting can be realized by spinning or suspending. A pending job is *ready* if it can be scheduled (a suspended job is not ready). We say that job J_i *makes progress* if a job that holds a resource for which J_i is waiting is scheduled and executing its critical section.

We denote J_i's k^{th} outermost request as $\mathcal{R}_{i,k}$, though we omit the request index k where it is inconsequential. We let N_i be the maximum number of outermost requests that J_i makes. The maximum duration of time that J_i executes (not counting suspensions and spinning) during its k^{th} outermost critical section is given by $L_{i,k}$.

Scheduling. We consider clustered-scheduled systems and job-level static-priority schedulers (we assume familiarity with these terms—recall that global and partitioned scheduling are special cases of clustered scheduling). We assume that there are $\frac{m}{c}$ clusters of size c each.

Each task has a *base priority* dependent upon the scheduling policy. A locking protocol can alter a job's priority such that it has a higher *effective priority*. Three such mechanisms, which we call *progress mechanisms*, exist to change a job's effective priority: *priority inheritance*, *priority boosting*, and *priority donation*. Priority boosting elevates a resource-holding job's priority to be higher than any base priority in the system so as to ensure that it is scheduled. Non-preemptive execution is an example of priority boosting. Under priority inheritance,



Processor
1
2
scheduled without resources
R
R
scheduled with resources R
job release
deadline
T
job suspended waiting for a token
@
job suspended waiting for ℓ_a
job waiting for ℓ_a by non-preemptively spinning
J_3
J_2
J_1
@
$\{\ell_a\}$
$\{\ell_a\}$
@
$\{\ell_a\}$
only s-aware pi-blocking
both s-oblivious and s-aware pi-blocking
0
5

Figure 2: Illustration (from [4]) of s-oblivious vs. s-aware analysis under global earliest-deadline-first scheduling on two processors. During $[2, 4)$, job J_3 is blocked, but there are m jobs with higher priority, so J_1 is not s-oblivious pi-blocked. However, because J_1 is also suspended, J_3 is s-aware pi-blocked. Intuitively, under s-oblivious analysis, the suspension time of higher-priority jobs is modeled as computation, but under s-aware analysis, it is not. (The legend applies to all figures.)

a resource-holding job's priority is elevated to that of the highest priority job waiting upon the held resource. Priority donation [5] is a hybrid of these two approaches: when a job J_d is released that would preempt a job J_i with an incomplete resource request, J_d is forced to suspend and donate its priority to J_i until J_i finishes its critical section. Priority boosting and priority donation both cause a type of blocking described later.

Blocking. We analyze locking protocols on the basis of *priority inversion blocking* (*pi-blocking*), i.e., the duration of time a job is blocked while a lower-priority job is running. Brandenburg and Anderson [4] defined two definitions of pi-blocking for tasks with suspensions, depending on whether schedulability analysis is *suspension-aware* (*s-aware*) (suspensions are considered) or *suspension-oblivious* (*s-oblivious*) (suspensions are modeled as computation).

Def. 1. Under **s-aware** analysis, a job J_i incurs *s-aware pi-blocking* if J_i is pending but not scheduled and fewer than c higher-priority jobs are **ready** in J_i's cluster.

Def. 2. Under **s-oblivious** analysis, a job J_i incurs *s-oblivious pi-blocking* if J_i is pending but not scheduled and fewer than c higher-priority jobs are **pending** in J_i's cluster.

The difference between s-oblivious and s-aware pi-blocking is demonstrated in Fig. 2. If waiting is realized by spinning, a different definition is required [2].

Def. 3. A job J_i incurs *spin-based blocking* if J_i is spinning (and thus scheduled) waiting for a resource.

For both spin- and suspension-based protocols, progress mechanisms such as non-preemptive spinning or priority donation can cause priority inversions for non-resource-using tasks. We call such blocking progress mechanism blocking, (*pm-blocking*), because it is the result of the progress mechanism; we note that often pm-

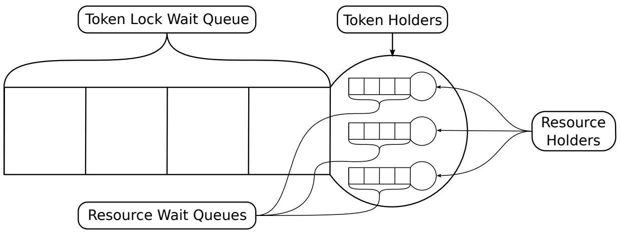


Figure 3: Components of the RNLP.

blocking happens upon job release, and thus has previously been termed release blocking. In contrast, we call pi-blocking (such as s-blocking) that occurs while a job has an incomplete resource request *request blocking*.

Analysis assumptions. We let L_{max} denote the maximum critical section length. In asymptotic analysis, we assume the number of processors m and tasks n to be variable, and all other variables constant, as in prior work [2, 4, 5, 12].

3 RNLP

The RNLP is composed of two components, a *token lock*, and a *request satisfaction mechanism* (*RSM*). When a job J_i requires a shared resource, it requests a token from the token lock. Once J_i has acquired a token, it issues a resource request to the RSM, which orders the satisfaction of resource requests. The overall architecture of the RNLP is shown in Fig. 3. Depending upon the system (clustered, partitioned, or globally scheduled), as well as the type of analysis being conducted (spin-based, s-oblivious, or s-aware), different tokens locks, number of tokens $\mathcal{T}$, and RSMs can be combined to form an efficient locking protocol.

The token lock is effectively a k-exclusion lock[1] that serves to limit the number of jobs that can have incomplete resource requests at a time. Therefore, existing k-exclusion locks can be employed as the token lock [5, 13] with $k = \mathcal{T}$.

As presented in [12], a single RSM controls access to all shared resources in the system. Associated with each resource ℓ_a is a resource queue RQ_a in the RSM that is ordered by the timestamp of token acquisition. This ordering is FIFO, but as seen below, a job that issues a nested request may "cut in line" to where it would have been had it issued the nested request at the time of token acquisition. Additionally, the RNLP prevents a request at the head of RQ_a from acquiring ℓ_a if another request with an earlier timestamp could issue a nested request for ℓ_a. These two properties effectively reserve spaces in all resource queues for the resources a job may request in the future. The non-greedy nature of these rules ensure that a request is never blocked by a request with a later timestamp (Lemma 1 of [12]), which results in efficient bounds on pi-blocking.

[1] k-exclusion generalizes mutual exclusion by allowing up to k simultaneous lock holders.

Analysis	Scheduler	$\mathcal{T}$	Progress Mechanism	pm-blocking	Request Blocking
spin	Any	m	Non-Preemptive Spinning	mL_{max}	$(m-1)L_{max}$
s-aware	Partitioned	n	Boosting	$(n-1)L_{max}$	$(n-1)L_{max}$
	Clustered	n	Boosting	$O(\phi \cdot n)$	$(n-1)L_{max}$
	Global†	n	Inheritance	$O(n)$	$(n-1)L_{max}$
s-oblivious	Partitioned	m	Donation	mL_{max}	$(m-1)L_{max}$
	Clustered	m	Donation	mL_{max}	$(m-1)L_{max}$
	Global	m	Donation	mL_{max}	$(m-1)L_{max}$
		m	Inheritance	0	$(2m-1)L_{max}$

† Applicable only under certain schedulers such as EDF and rate monotonic.

Table 1: Table adapted from [12], which gives the blocking behavior of different variants of the RNLP. L_{max} denotes the maximum critical section length. All listed protocols are asymptotically optimal except the case of clustered schedulers under s-aware analysis for which no asymptotically optimal locking protocol is known. ϕ is the ratio of the maximum to minimum period in the system.

The original rules of the RSM are given below.[2] In these rules, $\mathcal{L}_i$ denotes the set of resources that J_i may request in an outermost critical section under consideration (including nested requests). $\mathcal{L}_i$ can be specified at runtime when a job makes an outermost request, or defined implicitly via a partial ordering on allowable request nestings defined at build-time (in practice, this is commonly done by simply indexing resources).

Q1 When J_i acquires a token at time t for its k^{th} outermost critical section, the timestamp of token acquisition is recorded for the outermost request: $ts(\mathcal{R}_i) := t$. We assume a total order on such timestamps.

Q2 All jobs with requests in RQ_a wait with the possible exception of the job whose request is at the head of RQ_a.

Q3 A job J_i with an incomplete request $\mathcal{R}_i$ acquires ℓ_a when it is the head of RQ_a, and there is no request $\mathcal{R}_x$ with $ts(\mathcal{R}_x) < ts(\mathcal{R}_i)$ such that $\ell_a \in \mathcal{L}_x$.[3]

Q4 When a job J_i issues a request $\mathcal{R}_i$ for resource ℓ_a, $\mathcal{R}_i$ is enqueued in RQ_a in increasing timestamp order.[4]

Q5 When a job releases resource ℓ_a it is dequeued from RQ_a and the new head of RQ_a can gain access to ℓ_a, subject to Rule Q3.

Q6 When J_i completes its outermost critical section, it releases its token.

Example 1. To illustrate the key concepts of the RNLP, consider a globally scheduled earliest-deadline-first (EDF) system with $m = 2$ processors, $\mathcal{T} = 2$ tokens, and $q = 2$ resources, ℓ_a and ℓ_b, as seen in Fig. 4. Assume that a job that holds ℓ_a can make a nested request for ℓ_b, but not *vice versa*. At time $t = 0$, two jobs J_1 and J_2 are released, and later at time $t = 2$, jobs J_3 and J_4 are released. At time $t = 1$, J_1 makes a request for ℓ_a, and it thus acquires a token with $ts(\mathcal{R}_1) = 1$, and then immediately acquires ℓ_a. At time $t = 2$, J_2 requires ℓ_b, and it acquires a token with timestamp $ts(\mathcal{R}_2) = 2$. However,

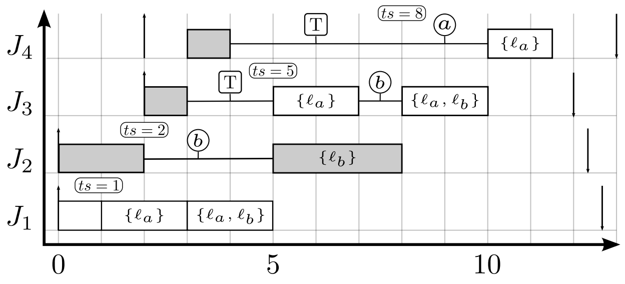


Figure 4: Illustration of Example 1. Note that during the interval $[5, 7)$ J_2 and J_3 are both scheduled under the RNLP, while a coarse-grained locking scheme would have disallowed such concurrency.

because J_1 could request ℓ_b in the future, J_2 suspends until time $t = 5$ by Rule Q3 when J_1 finishes its outermost critical section. While J_2 is suspended, J_3 requires ℓ_a at time $t = 3$. However, J_1 and J_2 hold the only two tokens, and thus J_3 must suspend and wait until J_1 releases its token at $t = 5$. At such time J_3 acquires ℓ_a despite having a later timestamp than J_2, because J_2 will never issue a request for ℓ_a. However, at time $t = 7$, when J_3 requires ℓ_b, it must suspend by Rule Q2 until time $t = 8$ when J_2 releases ℓ_b. Similarly, at time $t = 4$, J_4 requires ℓ_a but there is not an available token. J_4 suspends until time $t = 8$ when J_2 finishes its outermost critical section and releases its token. However, at time $t = 8$, ℓ_a is held, and thus J_4 must wait while holding a token for J_3 to release ℓ_a at time $t = 10$.

Table 1 summarizes the different variations of the original RNLP and their pi-blocking bounds [12]. We now turn our attention to modifications of the RNLP that resolve the issues raised in Sec. 1.

4 Dynamic Group Locks

Under fine-grained locking as provided by the original RNLP, a task may concurrently access multiple resources, but must acquire the locks on those resources individually. Under group locking, a task acquires a lock on an entire set of resources in one operation; however, this set may include far more resources than the task actually needs to access. In this section, we merge these two ways of supporting nesting in a mechanism we call *dynamic group*

[2] We adapted the notation from [12] for simplicity later. The two sets of rules are functionally identical.

[3] This rule was presented as Rule M1 in the online appendix of [12]. It generalizes the original Rule Q3.

[4] We assume that the acquisition of a token and subsequent enqueueing into the associated RQ occur atomically.

locks (*DGLs*). DGLs extend the notion of locking in the original RNLP by allowing a resource request to specify a *set* of resources to be locked. DGLs provide better concurrency than group locks, and lower system-call overheads than the original RNLP when the set of resources to lock in a nested fashion is known *a priori*. Also, DGLs do not alter the existing worst-case blocking bounds of the RNLP. Thus, the optimality of the RNLP is retained.

Note that DGLs can be supported in addition to nested locking, that is, tasks can issue nested DGL requests. Also, with the RNLP extended to support DGLs, individual nested requests can still be performed like before. Such nesting may be preferable to improve response times, as tasks are likely blocked by fewer requests. However, even if the set of resources that will actually be required is unknown—for example, when the resource access sequence is determined by executing conditional statements—DGLs can still be employed to request all resources that could be required, to reduce system-call overheads.

Rules. To enable the use of DGLs in the RNLP, we modify it as follows. When a job J_i requires a set of resources $\mathcal{D}_i$, it must first acquire a token, just as it would have under the original RNLP. Once J_i has acquired its token, its request is enqueued in the resource queue for each resource in $\{\mathrm{RQ}_a | \ \ell_a \in \mathcal{D}_i\}$. The DGL request is satisfied when it has acquired all resources in $\mathcal{D}_i$, at which point in time J_i is made ready. This can be expressed by replacing Rules Q3 and Rule Q4 with the following more general rules:

D1 A job J_i with an outstanding resource request $\mathcal{R}_i$ for a subset of resources $\mathcal{D}_i \subseteq \mathcal{L}$ acquires all resources in $\mathcal{D}_i$ when $\mathcal{R}_i$ is the head of every resource queue associated with a resource in $\mathcal{D}_i$, and there is no request $\mathcal{R}_x$ with $ts(\mathcal{R}_x) < ts(\mathcal{R}_i)$ for which there exists a resource $\ell_a \in \mathcal{D}_i \cap \mathcal{L}_x$.

D2 When a job J_i issues a request $\mathcal{R}_i$ for a set of resources $\mathcal{D}_i$, for every resource $a \in \mathcal{D}_i$, $\mathcal{R}_i$ is enqueued in RQ_a in timestamp order.

In an online appendix,[5] we prove that this modified version of the RNLP has the same worst-case blocking bounds as the original. Intuitively, such bounds do not change because a DGL request enqueues in multiple resource queues atomically when it is issued, instead of enqueueing in a single queue and essentially "reserving" slots in other queues for potential future nested requests. In the worst case, the set of blocking requests is the same in either case.

If all concurrent resource accesses in a system are supported by using DGLs, then the implementation of the RNLP can be greatly simplified. The timestamp-ordered queues become simple FIFO queues, and there is no need for jobs to "reserve" their position in any queue. This is due to the fact that all enqueueing due to one request is done atomically. Thus, in this case, not only is the number of system calls reduced, but the execution time of any one system call is likely lessened as well.

5 Reducing Short-On-Long Blocking

For ease of exposition, we explain how to reduce short-on-long blocking by considering the original RNLP as specified by Rules Q1–Q6. However, the modification to these rules explained below can easily be adapted to the DGL RNLP variant given previously.

In this section, we assume that each resource is either *short* or *long*,[6] similarly to [1]. All outermost critical sections during which only short resources are locked are themselves *short* and have a maximum length of L^s_{max}. All other requests are *long*, and have a maximum duration of $L^l_{max} > L^s_{max}$. We assume that a job holding a long resource can issue a nested request for a short resource, but not *vice versa*. This is a common assumption in practice so as to minimize the blocking time for the short resources. Also, we denote the subset of resources that are short (long) as $\mathcal{L}^s$ ($\mathcal{L}^l$).

In this model, the primary source of short-on-long request blocking under the RNLP is Rule Q3, which ensures that a request is never blocked by another request with a later timestamp. This rule effectively "reserves a slot" in a queue for all resources that could potentially be required by a task in the future. This can cause a request to be pi-blocked by requests for other resources, potentially of a different length. Nonetheless, Rule Q3 is sufficient to ensure optimal bounds on pi-blocking. In the presence of variant critical section lengths, this rule can be relaxed slightly to reduce short-on-long blocking.

The required relaxation is simple: a long request should not "reserve a slot" in any short-resource queue, even if it may issue nested requests for short resources in the future. This relaxation reduces short-on-long blocking, but at the expense of allowing long requests to be blocked by short requests with later timestamps. This increases the duration of pi-blocking for long requests, but only by a few short requests. However, we believe this to be an acceptable tradeoff. In this paper, we only consider this relaxation with waiting on short resources realized by spinning (as per the rules of the spin-based RSM [12]—see Table 1), and waiting for long resources realized by suspending, as recommended by [2]. However, the same idea can be applied if jobs wait for short resources by suspending.

To support both short and long resources within the RNLP without short-on-long blocking, we make two modifications. First, we employ two token locks, one for long resources and the other for short resources.[7] Second, we replace Rule Q3 with the two rules below.

Let $\mathcal{T} = \mathcal{T}_l + \mathcal{T}_s$ where $\mathcal{T}_s$ (resp. $\mathcal{T}_l$) is the number of tokens available to requests for short (resp. long) resources. A job that issues an outermost request that may

[5]Available at `http://www.cs.unc.edu/~bcw`.

[6]In the GPU use case mentioned in Sec. 1, critical sections with respect to GPU functional units are quite long and can be many orders of magnitude greater than short ones.

[7]This idea can be extended to create token locks for arbitrary subsets of resources at the expense of more verbose notation and analysis.

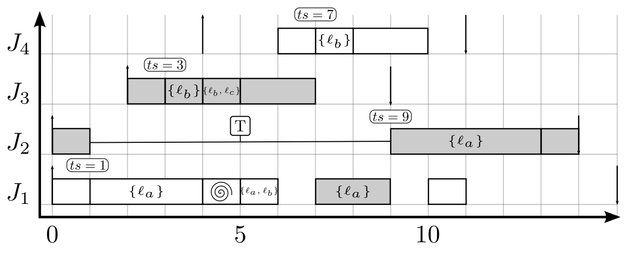


Figure 5: Illustration of Example 2 where $m = 2$ and $q = 3$.

include a long resource must compete for one of the $\mathcal{T}_l$ long tokens, while a job that issues an outermost request for exclusively short resources competes for one of the $\mathcal{T}_s$ short tokens. Also, let $C(\mathcal{R}_i, S)$ be the set of incomplete requests for at least one resource in S for which $\mathcal{R}_i$ contends. Importantly, a long request $\mathcal{R}_x$ that may issue a nested request for a short resource is only in $C(\mathcal{R}_i, \mathcal{L}^s)$ once it has issued its nested short request.

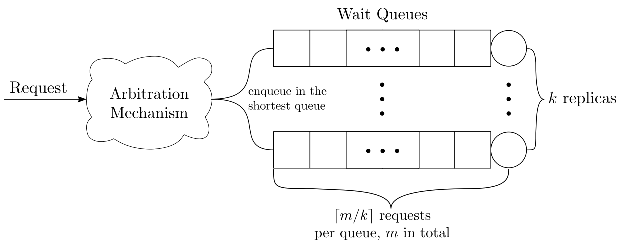
H1 A job J_i with an incomplete long request $\mathcal{R}_i$ for $\ell_a \in \mathcal{L}^l$ acquires ℓ_a when $\mathcal{R}_i$ is the head of RQ_a, and there is no incomplete long request $\mathcal{R}_x$ with $ts(\mathcal{R}_x) < ts(\mathcal{R}_i)$ and $\ell_a \in \mathcal{L}_x$.

H2 A job J_i with an incomplete short request $\mathcal{R}_i$ for $\ell_a \in \mathcal{L}^s$ acquires ℓ_a when $\mathcal{R}_i$ is the head of RQ_a, and there is no request $\mathcal{R}_x \in C(\mathcal{R}_i, \mathcal{L}_i)$ with $ts(\mathcal{R}_x) < ts(\mathcal{R}_i)$.

Example 2. Consider the two-processor system in Fig. 5, which is scheduled by global EDF with three resources, ℓ_a, ℓ_b, and ℓ_c, which can be locked in a nested fashion only in index order (ℓ_a before ℓ_b before ℓ_c). Assume that ℓ_a is long and ℓ_b and ℓ_c are short. (Thus, jobs suspend while waiting on ℓ_a, and spin while waiting on ℓ_b and ℓ_c.) Let $\mathcal{T}_s = 2$ and $\mathcal{T}_l = 1$.

Jobs J_1 and J_2 are released at time $t = 0$ with deadlines of 14 and 15, respectively. At time $t = 1$, both J_1 and J_2 need resource ℓ_a, J_1 acquires the only suspension token, and J_2 must suspend and wait for J_1 to release that token. At time $t = 2$, J_3 is released, and at time $t = 3$, J_3 issues a request for ℓ_b. J_3 then makes a nested request for ℓ_c at time $t = 4$, which is satisfied immediately. At time $t = 4$, J_1 issues a request for ℓ_b and spins until time $t = 5$ when J_3 releases ℓ_b and ℓ_c. Note that at time $t = 4$, J_1 is blocked by another job with a later timestamp. J_1 then executes non-preemptively (by the rules of the spin-based RSM [12]) with ℓ_a and ℓ_b, thereby pi-blocking J_4. When J_1 releases ℓ_b, J_4 executes because it has a sufficiently high priority and J_1 is no longer non-preemptive. When J_3 finishes at time $t = 7$, J_1 can resume its critical section. At time $t = 9$, J_1 releases ℓ_a, and J_2 finally acquires the suspension token, and acquires ℓ_a.

Note that under Rules Q1-Q6, under either spinning or suspending, both J_1 and J_2 would execute their critical sections before J_3 or J_4 could ever execute their critical sections. The combined length of the critical sections of J_1 and J_2 is 10, and thus Rule Q3 would not allow J_3 to acquire ℓ_b until $t = 11$, which is after its deadline. Under Rules Q1-Q6 this task set would be unschedulable. However, under Rule H2, a job accessing a long resource can



Wait Queues
Request
Arbitration Mechanism
enqueue in the shortest queue
k replicas
$\lceil m/k \rceil$ requests per queue, m in total

Figure 6: Figure illustrating the basic queue structure used in previous k-exclusion locking protocols. The arbitration mechanisms in these protocols behave similar to the token lock of the RNLP.

be blocked by a short request with a later timestamp, as is seen at time $t = 4$. Thus, the blocking term for the long resources is greater, but short requests are unaffected by long requests. Consequently, the task set is schedulable.

Detailed blocking analysis for this variant of the RNLP is given in an appendix. As seen there, the modifications above do not affect asymptotic blocking bounds, but eliminate short-on-long blocking.

6 Multi-Unit Multi-Resource Locking

In this section, we turn our attention to showing how to support replicated resources within the RNLP. We do this by leveraging recent work on asymptotically optimal real-time k-exclusion protocols [7, 8, 13]. Such protocols provide a limited form of replication: they enable requests to be performed on k replicas of a *single* resource. We desire to extend this functionality by allowing tasks to perform multiple requests simultaneously on replicas of *different* resources.

To motivate our proposed modifications to the RNLP, we consider three prior k-exclusion protocols, namely the O-KGLP [8], the K-FMLP [7], and the R^{2}DGLP [13], which function as depicted in Fig. 6. In these protocols, each replica is conceptually viewed as a distinct resource with its own queue. An "arbitration mechanism" (similar to our token lock) is used to limit the number of requests concurrently enqueued in these queues. In the case of s-aware (resp., s-oblivious) analysis, the arbitration mechanism is configured to allow up to n (resp., m) requests to be simultaneously enqueued. A "shortest queue" selection rule is used to determine the queue upon which a given request will be enqueued. This rule ensures that in the s-aware (resp., s-oblivious and spin-based) case, each queue can contain at most $\lceil n/k \rceil$ (resp., $\lceil m/k \rceil$) requests. From this, a pi-blocking bound of $O(n/k)$ (resp., $O(m/k)$) can be shown. Both bounds are asymptotically optimal.

Suppose now that we have two such replicated resources, as shown in Fig. 7, and that we wish to be able to support requests that involve accessing two replicas, one per resource, simultaneously. If the enqueueing associated with such a request is done by the arbitration mechanism atomically, then this is simple to do: as a result of processing the request, it is enqueued onto the shortest queue associated with each resource *at the same time*. This simple generalization of the aforementioned k-exclusion al-

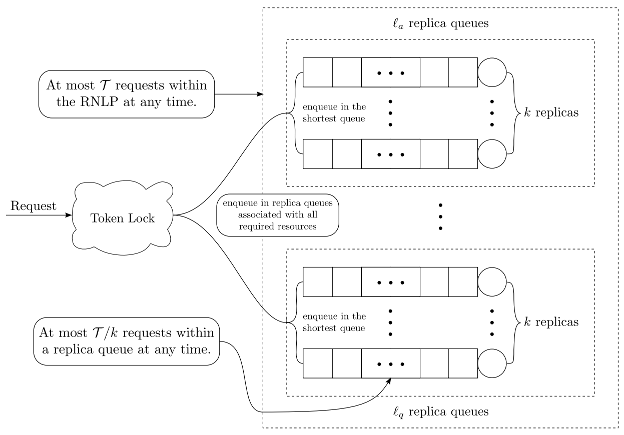

ℓ_a replica queues
At most $\mathcal{T}$ requests within the RNLP at any time.
enqueue in the shortest queue
k replicas
Request
Token Lock
enqueue in replica queues associated with all required resources
enqueue in the shortest queue
k replicas
At most $\mathcal{T}/k$ requests within a replica queue at any time.
ℓ_q replica queues

Figure 7: Figure illustrating how DGL can be used to request replicas of different resources.

gorithms retains their optimal pi-blocking bounds.

Note that the functionality just described is provided by DGLs. Thus, to support multiple replicas when simultaneous lock holding is done only via DGLs (and not nesting), we merely need to treat each replica as a single resource and use a "shortest queue" rule in determining the replica queues in which to place a request. If each resource is replicated at least k times then it is straightforward to show that the earlier-stated pi-blocking bounds of $O(m/k)$ and $O(n/k)$ for s-oblivious and s-aware analysis, respectively, still apply. As before, both bounds are asymptotically optimal.

If simultaneous lock holding is done via nesting, then the situation is a bit more complicated. This is due to the RNLP's conservative resource acquisition rule (Rule Q3), which enables a request with a lower timestamp to effectively "reserve" its place in line within any queue of any resource it may request in the future. This rule causes problems with replicated resources. Consider again Fig. 7. Consider an outermost request $\mathcal{R}_i$ for ℓ_a that *may* make a nested request for ℓ_q. Which replica queue for ℓ_q should hold its "reservation?" If a specific queue is chosen by the "shortest queue" rule when $\mathcal{R}_i$ receives its timestamp, and if $\mathcal{R}_i$ does indeed generate a nested request for ℓ_q later, then the earlier-selected queue may not still be the shortest for ℓ_q when the nested request is made. If a queue is not chosen until the nested request is made, then since $\mathcal{R}_i$ had no "reservation" in any queue of ℓ_q until then, it could be the case that requests with later timestamps hold all replicas of ℓ_q when the nested request is made. This violates a key invariant of the RNLP.

Our solution is to require $\mathcal{R}_i$ to conceptually place a reservation in the shortest replica queue for each resource that may be required in the future. The idea is to enact a "DGL-like" request for $\mathcal{R}_i$ when it receives a token that enqueues a "placeholder" request for $\mathcal{R}_i$ on one replica queue, determined by the "shortest queue" rule, for each resource it *may* access. Such a placeholder can later be canceled if it is known that the corresponding request will not be made. Thus, as before, nesting and DGLs are equivalent from the perspective of worst-case asymptotic pi-blocking.

7 Experimental Results

Next we present an experimental evaluation of fine-grained locking via the RNLP through a *schedulability* study. In this study, we evaluated the schedulability of randomly generated task systems, and report the fraction that are schedulable. These experiments were designed to depict the effect of blocking bounds on schedulability, and therefore do not include overheads. A full overhead-aware schedulability study is deferred to future work, though we note the presented techniques have been implemented and been proven useful in the context of the aforementioned GPU [9] and shared-cache [14] use cases.

We randomly generated task systems using a similar experimental design as previous studies (e.g., [4]). We assume that tasks are partitioned onto $m = 8$ processors, and scheduled with EDF priorities. We also assume that all tasks have implicit deadlines ($d_i = p_i$). We generated task systems with total system utilizations in $\{0.1, 0.2, \ldots, 8.0\}$. The per-task utilizations where chosen uniformly from the range $[0.1, 0.4]$ or $[0.5, 0.9]$, denoted, *medium* or *heavy*, respectively. The period of each task was chosen uniformly from either $[3, 33]\,\mathrm{ms}$ (*short*) or $[50, 250]\,\mathrm{ms}$ (*long*). All tasks were assumed to access $N \in \{2, 4, 8\}$ of 16 shared resources. The duration of each critical section was exponentially distributed with a mean of either $10\,\mu\mathrm{s}$ (*small*) or $1000\,\mu\mathrm{s}$ (*large*).

For each generated task set, we evaluated hard real-time (HRT) schedulability under four different locking protocols, two coarse-grained protocols, the mutex OMLP and the clustered k-exclusion variant of the OMLP [5] (denoted CK-OMLP), and two fine-grained protocols, the RNLP [12] and the k-exclusion RNLP variant presented herein (denoted K-RNLP). For the fine-grained protocols, additional analysis optimizations where included that are based on evaluations of possible transitive blocking relationships.[8] We present a subset of our generated graphs in Figs. 8-10, in which all critical section lengths are long.

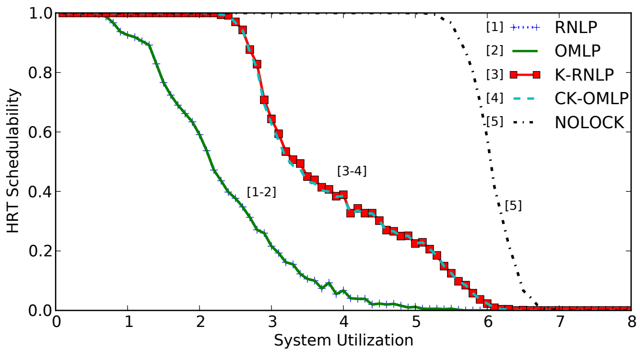
Obs. 1. Schedulability is no worse using a fine-grained locking protocol than a similar coarse-grained one.

This observation is supported by Fig. 8, which depicts the schedulability of two different system configurations. Inset (a) depicts a system in which fine-grained locking provides little if any schedulability benefit over coarse-grained locking for either mutex or k-exclusion locks. Inset (b), on the other hand, depicts a system in which fine-grained locking provides more significant schedulability benefits owing to the additional analysis optimizations. We note that the blocking bounds for the coarse-grained locking protocols upper bound the worst-case blocking for the fine-grained protocols, and thus the fine-grained protocols will perform no worse than the coarse-grained ones.

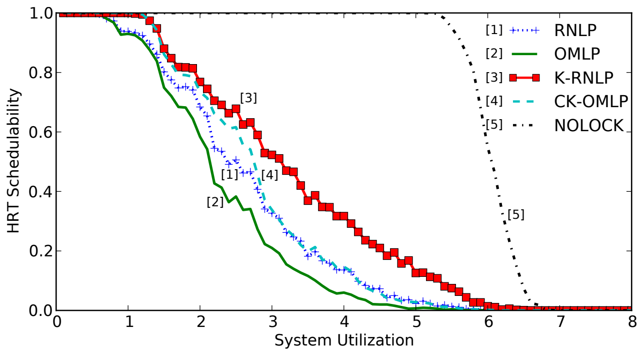
Obs. 2. Resource replication improves schedulability.

This observation is supported by Fig. 9, which depicts the schedulability of a given system under different de-

[8]Tighter analysis than that employed in these experiments is possible using an exponential-time algorithm. Such analysis, while perhaps computationally tractable for a single task system, is intractable when evaluating hundreds of thousands of task systems.



(a) $k = 4$, short periods, $N = 8$, heavy per-task utilizations.



(b) $k = 2$, short periods, $N = 2$, heavy per-task utilizations.

Figure 8: Sample schedulability results. Inset (a) demonstrates that fine-grained nesting, in some cases provides little if any advantage over coarse-grained nesting. Inset (b) demonstrates that in other cases, fine-grained nesting can provide more significant schedulability benefits over coarse-grained nesting.

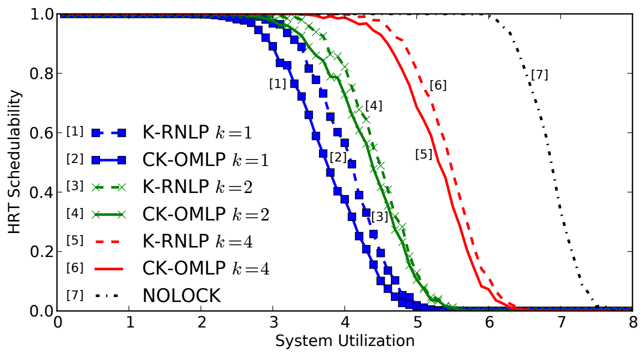


Figure 9: Illustration of the improved schedulability made possible with a higher degree of resource replication. In this figure, periods are long, per-task utilizations are medium, and $N = 2$.

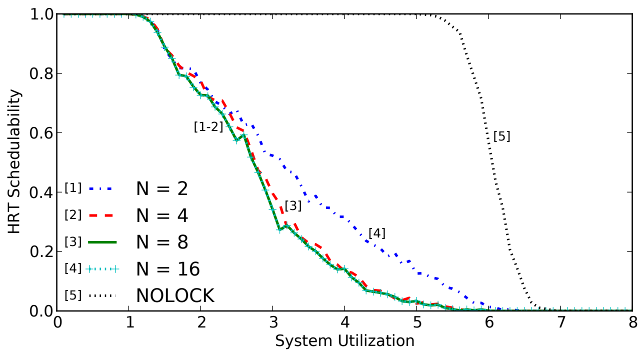


Figure 10: Illustration of the effect of the number of resources accessed within an outermost critical section on schedulability. In this figure, periods are short, per-task utilizations are heavy, and $k = 2$.

grees of resource replication. When resources are more highly replicated, more requests can be satisfied concurrently, which decreases blocking bounds. As described in Sec. 6, the reduced blocking made possible by resource replication can be reflected in the worst-case blocking bound, which is $O(m/k)$. This improved blocking bound results in improved schedulability, as is seen in Fig. 9.

Obs. 3. Fine-grained locking improves schedulability over coarse-grained locking most when the number of resources accessed within an outermost critical section is small.

This observation is corroborated by Fig. 10, which depicts the schedulability of a given system with tasks requesting different numbers of resources, N. In that particular system, the schedulability when $N = 2$ is considerably better than when $N > 2$, but the benefits of fine-grained nesting diminish with larger N. This is because when the number of resources accessed within a critical section is small, fine-grained locking is more likely to allow non-conflicting requests, which would have been serialized under coarse-grained locking, to be satisfied concurrently. In many cases, as is seen in Fig. 10, this parallelism can be reflected in the blocking analysis (though it does not effect blocking bounds asymptotically). Note also the number of resources accessed within an outermost critical section are often small in practice [3]. Thus, the cases in which fine-grained locking performs best are the most common in practice.

From these results, we conclude that fine-grained locking protocols offer improved schedulability over coarse-grained ones. Furthermore, we note that even in cases in which fine-grained locking provides no analytical benefit, it is still preferable at in practice as it may lead to improved response times and therefore safety margins and responsiveness. In the future, we plan to implement these protocols, measure overheads, and conduct an overhead-aware schedulability study.

8 Conclusions

We have presented several extensions to the RNLP [12] that address issues of practical concern that arise when attempting to support nested resource requests in real-time multiprocessor systems in a fine-grained way. First, we introduced dynamic group locks (DGLs) to reduce system-call overhead when the set of resources to lock is known *a priori*. With support for DGLs added, the RNLP generalizes standard group locking by allowing groups of resources to be atomically locked dynamically and by allowing such locks to be nested.

Second, we addressed the problem of short-on-long blocking, which occurs when a short resource request is blocked by a long resource request. This is a potential problem, for example, when a single synchronization protocol is used to control access to both I/O devices as well as shared memory objects. We generalized the RNLP by biasing its rules to favor short requests over long ones. This eliminates short-on-long blocking at the expense of creating a modest amount of long-on-short blocking. This new variant of the RNLP is also of interest because it allows both spin- and suspension-based waiting to be used in the same synchronization protocol.

Finally, we showed how to incorporate replicated resources within the RNLP. Viewing different resources as replicas of a single resource is useful when a task only requires access to *some* replica and not a particular one. We also conducted a schedulability study of this RNLP, which showed that fine-grained locking offered improved schedulability over coarse-grained locking in many cases.

To simplify the presentation, we have for the most part considered these various extensions separately from one another. However, they can all be combined into a single extended RNLP with no adverse impact on asymptotic pi-blocking bounds.

When designing a system that employs these techniques, there are many design decisions that can be made to make the system schedulable, or, in a soft real-time system, improve response times. Resources can be grouped and marked as short or long, resource replicas can be determined, tasks can be partitioned across clusters of processors in such a way so as to minimize blocking and improve schedulability, etc. In future work, we plan to investigate algorithms to automate this design process so as to improve the chance of a system being schedulable. Additionally, we plan to implement these techniques and evaluate them empirically.

References

[1] A. Block, H. Leontyev, B.B. Brandenburg, and J.H. Anderson. A flexible real-time locking protocol for multiprocessors. In *RTCSA '07*, pages 47–56, Aug. 2007.

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

[3] B.B. Brandenburg and J.H. Anderson. Feather-trace: A light-weight event tracing toolkit. In *OSPERT '07*, pages 61–70, 2007.

[4] B.B. Brandenburg and J.H. Anderson. Optimality results for multiprocessor real-time locking. In *RTSS '10*, pages 49–60, 2010.

[5] B.B. Brandenburg and J.H. Anderson. Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In *EMSOFT '11*, pages 69–78, Sep. 2011.

[6] A Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In *RTSS '09*, pages 377–386, 2009.

[7] G.A. Elliott and J.H. Anderson. Robust real-time multiprocessor interrupt handling motivated by GPUs. In *ECRTS '12*.

[8] G.A. Elliott and J.H. Anderson. An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems. In *RTNS '11*, pages 15–24, Sep. 2011.

[9] G.A Elliott, B.C Ward, and J.H. Anderson. GPUSync: A framework for real-time GPU management. (in submission).

[10] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *RTSS '09*, 2009.

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

[12] B.C. Ward and J.H. Anderson. Supporting nested locking in multiprocessor real-time systems. In *ECRTS '12*.

[13] B.C. Ward, G.A Elliott, and J.H. Anderson. Replica-request priority donation: A real-time progress mechanism for global locking protocols. In *RTCSA '12*.

[14] B.C. Ward, J.L. Herman, C.J Kenna, and J.H Anderson. Making shared caches more predictable on multicore platforms. In *ECRTS '13*.

A Short-on-Long Analysis

(Recall that, in describing how to eliminate short-on-long blocking, we assumed that DGLs are not also supported, to simplify the presentation. We assume that here as well.) For analysis, we must have additional information about inner critical sections. We define a *nested outermost short request* to be a nested request for a short resource that is outermost with respect to short resources. Additionally, if $\mathcal{R}_i$ is a long request, then we let $N^s_{i,k}$ denote the number of nested short request within it. When analyzing the blocking behavior of short resources, we must consider that a short request can be blocked by all $N^s_{i,k}$ short requests of J_i. Additionally, when analyzing the blocking behavior of a long resource request, we must account for the fact that within each long outermost critical section, a job can be blocked by up to $N^s_{i,k}$ short requests with later timestamps.

Let L^l_{max} (L^s_{max}) be the maximum critical section length for a long (short) outermost request. Additionally, let N^s_{max} be the maximum number of nested outermost short requests a job makes within a long outermost critical section.

Before conducting rigorous analysis, we must first redefine the definitions of direct and indirect blocking so as to account for the blocking behavior of Rules H1 and H2. These rules allow for a job to be blocked by a job holding a short resource with a later timestamp. Thus, we must update the definition of direct blocking in the case that a job is blocked by a later timestamp short request. Let $h(\ell_a, t)$ be the request holding ℓ_a at time t and let $w(\mathcal{R}_i, t)$ be the resource for which $\mathcal{R}_i$ is waiting at time t. Also, assume that $DB(\mathcal{R}_i)$ is the original definition of direct blocking from [12]. If $ts(\mathcal{R}_i) < ts(h(w(\mathcal{R}_i, t), t))$, then the new definition of direct blocking, denoted $DB'(\mathcal{R}_i)$ is equal to $DB(\mathcal{R}_i, t)$, otherwise, $DB'(\mathcal{R}_i, t) = h(w(\mathcal{R}_i, t), t) \cup DB(h(w(\mathcal{R}_i, t), t))$. Additionally, Rules H1 and H2 change the definitions of indirect blocking. Under these rules, the expression for indirect blocking is dependent upon whether a job is waiting for a short or long resource, we denote these cases as

$IB^s(\mathcal{R}_i, t)$ and $IB^l(\mathcal{R}_i, t)$ respectively.

$$\begin{aligned} IB^s(\mathcal{R}_i, t) = \{\mathcal{R}_x \in RQ_a \mid\ & \ell_a \in \mathcal{L}^s, \\ & w(\mathcal{R}_i, t) \in \mathcal{L}_x \wedge \\ & ts(\mathcal{R}_x) < ts(\mathcal{R}_i)\} \\ IB^l(\mathcal{R}_i, t) = \{\mathcal{R}_x \in RQ_a \mid\ & \ell_a \in \mathcal{L}^l, \\ & w(\mathcal{R}_i, t) \in \mathcal{L}_x \wedge \\ & ts(\mathcal{R}_x) < ts(\mathcal{R}_i)\} \end{aligned}$$

Lemma 1. *A job is never blocked by a long critical section of a job with a later timestamp.*

Proof. Assume J_x is executing an outermost critical section that is long while holding ℓ_a. Then $\mathcal{R}_x$ has been satisfied. Now consider a request $\mathcal{R}_i$ that $\mathcal{R}_x$ blocks. If $ts(\mathcal{R}_i) < ts(\mathcal{R}_x)$, then Rule H1 or H2 would have prevented $\mathcal{R}_x$ from being satisfied, because $\mathcal{R}_i$ could request ℓ_a in the future. Thus, a job can never be blocked by a long request with a later timestamp. □

Lemma 2. *Within the long outermost critical section of $\mathcal{R}_i$, $\mathcal{R}_i$ can be blocked by at most one outermost short request with a later timestamp for each short request nested within $\mathcal{R}_i$.*

Proof. Rule H2 allows an outstanding outermost request for a short resource ℓ_s to be satisfied even if an incomplete long request $\mathcal{R}_i$ with a later timestamp may request ℓ_s in the future. Thus, each time such a request $\mathcal{R}_i$ makes a nested short request, it could be blocked by up to one (but no more) outermost short request with a later timestamp. □

Lemma 3. *An outermost long request $\mathcal{R}_i$ can be request blocked by the RSM for a total duration of at most $(\mathcal{T}_l N^s_{max} + \mathcal{T}_s)L^s_{max} + (\mathcal{T}_l - 1)L^l_{max}$.*

Proof. By Lemma 1, $\mathcal{R}_i$ can be blocked by three types of requests: long requests with earlier timestamps, short requests with earlier timestamps, and short requests with later timestamps. We next quantify each of these blocking terms.

There can be at most $\mathcal{T}_l - 1$ long token-holding requests with earlier timestamps than $\mathcal{R}_i$, each of which executes for up to L^l_{max} time. This accounts for $(\mathcal{T}_l - 1)L^l_{max}$ blocking. It is also possible for $\mathcal{R}_i$ to issue nested requests for short resources and be blocked by $\mathcal{T}_s$ short requests with earlier timestamps, resulting in up to $\mathcal{T}_s L^s_{max}$ units of blocking from short requests with earlier timestamps. Finally, by Lemma 2, $\mathcal{R}_i$ and up to $\mathcal{T}_l - 1$ other long requests with earlier timestamps that block $\mathcal{R}_i$ may each be blocked by an outermost short request with a larger timestamp for each nested short request they make. This creates up to $\mathcal{T}_l N^s_{max} L^s_{max}$ additional blocking due to short requests with later timestamps. □

Thus, a job can *hold* the long token for at most $(\mathcal{T}_l N^s_{max} + \mathcal{T}_s)L^s_{max} + \mathcal{T}_l L^l_{max}$ time.

Lemma 4. *An outermost short request $\mathcal{R}_i$ can be request blocked by the RSM for a total duration of at most $(\min(m, \mathcal{T}) - 1)L^s_{max}$.*

Proof. (From Table 1, jobs that are waiting by spinning are assumed to be priority boosted.) While there can be at most $\mathcal{T}_s$ jobs holding short tokens, the $\mathcal{T}_l$ jobs holding long tokens can additionally make nested resource requests, resulting in a total of $\mathcal{T}$ jobs that can have incomplete short requests at a time. However, because waiting is realized by spinning for short resources and there can be at most m spinning jobs, a job can be blocked by at most $m-1$ short requests. Thus, the maximum duration of request blocking for $\mathcal{R}_i$ is given by $(\min(m, \mathcal{T}) - 1)L^s_{max}$. □

Thus, a job can *hold* the short token for at most $\min(m, \mathcal{T})L^s_{max}$ time.

Lemmas 3 and 4 only quantify how long a request can be request blocked by the RSM. Next we consider the total duration of request blocking a job can experience by incorporating the duration of blocking in the token lock. We first consider total s-oblivious request blocking.

Theorem 1. *Under s-oblivious analysis, an outermost long request $\mathcal{R}_i$ can be request blocked for a total duration of at most*

$$2\left\lceil \frac{m}{\mathcal{T}_l} \right\rceil \left((\mathcal{T}_l N^s_{max} + \mathcal{T}_s)L^s_{max} + \mathcal{T}_l L^l_{max}\right) - L^l_{max}$$

assuming a token lock with a worst-case blocking term given by $(2\lceil \frac{m}{\mathcal{T}_l} \rceil - 1)L_{max}$ (where L_{max} is w.r.t. to the critical section of the k-exclusion lock) is employed (such as the CK-OMLP [5] or the R^{2}DGLP [12]).

Proof. $\mathcal{R}_i$ is delayed by the request blocking it experiences once it receives a token, which is given by the bound in Lemma 3, and also by the token hold time of every request that may receive a token before it. The token hold time is L^s_{max} greater than the bound in Lemma 3, and by the stated assumption concerning the token lock, there are at most $2\lceil \frac{m}{\mathcal{T}_l} \rceil - 1$ such preceding requests in total to consider. □

Theorem 2. *An outermost short request $\mathcal{R}_i$ can be request blocked for a total duration of at most*

$$\max(0, \min(m, \mathcal{T})L^s_{max}(m - \mathcal{T}_s - 1)) + (\min(m, \mathcal{T}) - 1)L^s_{max}.$$

Proof. Given that spinning jobs are boosted, at most m short requests can execute concurrently. Hence, the token queue for short requests is of length at most $m - \mathcal{T}_s$. The rest of the proof is similar to that of Theorem 1: we have to account for the token hold time of up to $m - \mathcal{T}_s - 1$ requests and the request blocking experienced by $\mathcal{R}_i$ while it holds the token. By Lemma 4, the former is at most $\max(0, \min(m, \mathcal{T})L^s_{max}(m - \mathcal{T}_s - 1))$ and the latter is at most $(\max(m, \mathcal{T}) - 1)L^s_{max}$. □

According to Theorem 2, it is likely best in practice to set $\mathcal{T}_s = m$, for in this case, the total worst-case request blocking per outermost short request is merely $(m - 1)L^s_{max}$, which is independent of $\mathcal{T}_l$.