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 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 have asymptotically optimal blocking behavior.

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:

1. 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.

2. 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.

3. In applications in which resources are replicated (e.g., GPUs), viewing each replica as a distinct resource may cause unnecessary blocking if a task merely requires ac-
cess to some replica and not a specific one. The original RNLP [12] 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 by either spinning or suspending; thus, we allow both waiting mechanisms to be used in the same protocol. In the original RNLP [12], 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 τ = {T1, . . . , Tn} that execute on m processors. We denote the kth job (invocation) of the ith task as Ji,k, though we often omit the job index k if it is insignificant. Each task Ti is characterized by a worst-case execution time ci, minimum job separation pi, and relative deadline di. For simplicity, we assume implicit deadlines (di = pi), and that every job must complete before its deadline (no tardiness). We say that a released job is pending until it finishes its execution.

Resources. We consider a system that contains q shared resources L = {ℓ1, . . . , ℓq}. 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 Ji makes progress if a job that holds a resource for which Ji is waiting is scheduled and executing its critical section.

We denote Ji’s kth outermost request as Rk,i,k, though we omit the request index k where it is inconsequential. We let Ni be the maximum number of outermost requests that Ji makes. The maximum duration of time that Ji executes (not counting suspensions and spinning) during its kth outermost critical section is given by Li,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 q clusters of c processors 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, 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 Ji is released that would preempt a job Jd with an incomplete resource request, Jd is forced
suspensions are modeled as computation. pensions are considered) or suspension-aware schedulability analysis is blocking for tasks with suspensions, depending on whether Brandenburg and Anderson [4] defined two definitions of π-time a job is blocked while a lower-priority job is running. Priority boosting and priority donation both priority inversion blocking. We analyze locking protocols on the basis of priority inversion blocking (π-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 π-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), but under suspension analysis, it is not. (The legend applies to all figures.)

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.)

to suspend and donate its priority to \( J_1 \) until \( J_1 \) 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 (π-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 π-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 \( J \) incurs s-aware π-blocking if \( J \) is pending but not scheduled and fewer than \( c \) higher-priority jobs are ready in \( J \)'s cluster.

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

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

**Def. 3.** A job \( J \) incurs spin-based blocking if \( J \) 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-blocking happens upon job release, and thus has previously been termed release blocking. In contrast, we call π-blocking (such as s-blocking) that occurs while a job has an incomplete resource request request blocking.

**Analysis assumptions.** We let \( L_{\text{max}} \) 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 \( J \) requires a shared resource, it requests a token from the token lock. Once \( J \) 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 \( T \), 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 \( k = T \).

As presented in [12], a single RSM controls access to all shared resources in the system. Associated with each resource \( \ell_a \) is a resource queue \( \text{RQ}_a \) 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 \( \text{RQ}_a \) from acquiring \( \ell_a \) if another request with an earlier timestamp could issue a nested request for \( \ell_a \). 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.

The original rules of the RSM are given below.\(^2\) In these rules, \( L \) denotes the set of resources that \( J \) may request in an outermost critical section under consideration (including nested requests). \( L \) can be specified at run-time when a job makes an outermost request, or defined implicitly via a partial ordering on allowable request nestings defined off-

---

\(^1\)\( k \)-exclusion generalizes mutual exclusion by allowing up to \( k \) simultaneous lock holders.

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

---

**Figure 2**

![Figure 2: Illustration (from [4]) of s-oblivious vs. s-aware analysis under global earliest-deadline-first scheduling on two processors.](image)

**Figure 3**

![Figure 3: Components of the RNLP.](image)
Table 1: Table adapted from [12], which gives the blocking behavior of different variants of the RNLP. \(L_{\text{max}}\) 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. \(\phi\) is the ratio of the maximum to minimum period in the system.

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Scheduler</th>
<th>(\mathcal{f})</th>
<th>Progress Mechanism</th>
<th>pin-blocking</th>
<th>Request Blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>spin</td>
<td>Any</td>
<td>(m)</td>
<td>Non-Preemptive Spinning</td>
<td>(mL_{\text{max}})</td>
<td>((m - 1)L_{\text{max}})</td>
</tr>
<tr>
<td>s-aware</td>
<td>Partitioned</td>
<td>(n)</td>
<td>Boosting</td>
<td>((n - 1)L_{\text{max}})</td>
<td>((n - 1)L_{\text{max}})</td>
</tr>
<tr>
<td></td>
<td>Clusters</td>
<td>(n)</td>
<td>Boosting</td>
<td>(O(\phi \cdot n))</td>
<td>((n - 1)L_{\text{max}})</td>
</tr>
<tr>
<td></td>
<td>Global†</td>
<td>(n)</td>
<td>Inheritance</td>
<td>(O(n))</td>
<td>((n - 1)L_{\text{max}})</td>
</tr>
<tr>
<td>s-oblivious</td>
<td>Partitioned</td>
<td>(m)</td>
<td>Donation</td>
<td>(mL_{\text{max}})</td>
<td>((m - 1)L_{\text{max}})</td>
</tr>
<tr>
<td></td>
<td>Clusters</td>
<td>(m)</td>
<td>Donation</td>
<td>(mL_{\text{max}})</td>
<td>((m - 1)L_{\text{max}})</td>
</tr>
<tr>
<td></td>
<td>Global</td>
<td>(m)</td>
<td>Donation</td>
<td>(mL_{\text{max}})</td>
<td>((m - 1)L_{\text{max}})</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(m)</td>
<td>Inheritance</td>
<td>0</td>
<td>((2m - 1)L_{\text{max}})</td>
</tr>
</tbody>
</table>

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

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.

Example 1. To illustrate the key concepts of the RNLP, consider a globally scheduled earliest-deadline-first (EDF) system with \(m = 2\) processors, \(T = 2\) tokens, and \(q = 2\) resources, \(\ell_a\) and \(\ell_b\), as seen in Fig. 4. Assume that a job that holds \(\ell_a\) can make a nested request for \(\ell_b\), 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 \(\ell_a\), and it thus acquires a token with \(ts(\mathcal{R}_1) = 1\), and then immediately acquires \(\ell_a\). At time \(t = 2\), \(J_2\) requires \(\ell_b\), and it acquires a token with timestamp \(ts(\mathcal{R}_2) = 2\). However, because \(J_1\) could request \(\ell_b\) in the future, \(J_1\) suspends until time \(t = 5\) by Rule Q3 when \(J_1\) finishes its outermost critical section. While \(J_2\) is suspended, \(J_3\) requires \(\ell_a\) 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 \(\ell_a\) despite having a later timestamp than \(J_2\), because \(J_2\) will never issue a request for \(\ell_a\). However, at time \(t = 7\), when \(J_3\) requires \(\ell_b\), it must suspend by Rule Q2 until time \(t = 8\) when \(J_2\) releases \(\ell_b\). Similarly, at time \(t = 4\), \(J_4\) requires \(\ell_a\) 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 \(t = 8\), \(\ell_a\) is held, and thus \(J_4\) must wait while holding a token for \(J_3\) to release \(\ell_a\) at time \(t = 10\).

Table 1 summarizes the different variations of the original RNLP and their pin-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 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 $D_i$, 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 $\{RQ_a | \ell_a \in D_i\}$. The DGL request is satisfied when it has acquired all resources in $D_i$, 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 $R_a$ for a subset of resources $D_i \subseteq L$ acquires all resources in $D_i$ when $R_a$ is the head of every resource queue associated with a resource in $D_i$, and there is no request $R_x$ with $ts(R_x) < ts(R_a)$ for which there exists a resource $\ell_x \in D_i \cap L$.  

**D2** When a job $J_i$ issues a request $R_a$ for a set of resources $D_i$, for every resource $a \in D_i$, $R_a$ is enqueued in $RQ_a$ in timestamp order.

In Appendix A, 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 enqueuing 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 enqueuing 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, similarly to [1]. All outermost critical sections during which only short resources are locked are themselves short and have a maximum length of $L^\text{max}_s$. All other requests are long, and have a maximum duration of $L^\text{max}_l > L^\text{max}_s$. 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 $L^s$ ($L^l$).

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. Second, we replace Rule Q3 with the two rules below.

**H1** A job $J_i$ with an incomplete long request $R_a$ for $\ell_a \in L^l$ acquires $\ell_a$ when $R_a$ is the head of $RQ_a$, and there is no incomplete long request $R_x$ with $ts(R_x) < ts(R_a)$ and $\ell_x \in L^l$.

**H2** A job $J_i$ with an incomplete short request $R_a$ for $\ell_a \in L^s$ acquires $\ell_a$ when $R_a$ is the head of $RQ_a$, and there is no incomplete request for a short resource is only in $C(R_a, L^s)$ once it has issued its nested short request.

In the GPU use case mentioned in Sec. 1, critical sections

---

3In the GPU use case mentioned in Sec. 1, critical sections

---

6This idea can be extended to create token locks for arbitrary subsets of resources at the expense of more verbose notation and analysis.
is no request \( R_x \in C(R_i, L_i) \) with \( ts(R_x) < ts(R_y) \).

**Example 2.** Consider the two-processor system in Fig. 5, which is scheduled by global EDF with three resources, \( \ell_a \), \( \ell_b \), and \( \ell_c \), which can be locked in a nested fashion only in index order (\( \ell_a \) before \( \ell_b \) before \( \ell_c \)). Assume that \( \ell_a \) is long and \( \ell_b \) and \( \ell_c \) are short. (Thus, jobs suspend while waiting on \( \ell_a \), and spin while waiting on \( \ell_b \) and \( \ell_c \).) Let \( T_s = 2 \) and \( T_f = 1 \).

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 \( \ell_a \). \( J_1 \) acquires the only suspension token, and \( J_2 \) must suspend and wait for \( J_1 \) to release that token. At time \( t = 2 \), \( J_1 \) is released, and at time \( t = 3 \), \( J_3 \) issues a request for \( \ell_b \). \( J_3 \) then makes a nested request for \( \ell_c \) at time \( t = 4 \), which is satisfied immediately. At time \( t = 4 \), \( J_1 \) issues a request for \( \ell_b \) and spins until time \( t = 5 \) when \( J_3 \) releases \( \ell_b \) and \( \ell_c \). 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 \( \ell_a \) and \( \ell_b \), thereby pi-blocking \( J_1 \). When \( J_1 \) releases \( \ell_b \), \( J_4 \) 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 \( \ell_a \), and \( J_2 \) finally acquires the suspension token, and acquires \( \ell_a \).

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 \( \ell_b \) 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 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 Appendix B. 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 R2DGLP [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 \( \lfloor n / k \rfloor \) (resp., \( \lfloor m / k \rfloor \)) 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 enqueuing 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 algorithms 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 \( R_i \) for \( \ell_q \) that may make a nested request for \( \ell_q \). Which replica queue for \( \ell_q \) should hold its “reservation”? If a specific queue is chosen by the “shortest queue” rule when \( R_i \) receives its timestamp, and if \( R_i \) does indeed generate a nested request for \( \ell_q \) later, then the earlier-selected queue may not still be the shortest for \( \ell_q \) when the nested request is made. If a queue is not chosen until the nested request is made, then since \( R_i \) had no “reservation” in any queue of \( \ell_q \) until then, it could be the case that requests with later timestamps hold all replicas of \( \ell_q \) when the nested request is made. This violates a key invariant of the RNLP.

Our solution is to require \( R_i \) 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 \( R_i \) when it receives a token that enqueues a “placeholder” request for \( R_i \) 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 \( \{0.1, 0.2, \ldots, 0.8\} \). 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 \([3, 33] \) ms (short) or \([50, 250] \) 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 \(10 \mu \text{s} \) (small) or \(1000 \mu \text{s} \) (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). We also evaluated the schedulability of the task system assuming no critical sections (denoted NOLOCK). For the fine-grained protocols, additional analysis optimizations where included that are based on evaluations of possible transitive blocking relationships.\(^7\) We present a subset of our generated graphs in Figs. 8-10, in which all critical section lengths are long.

\( \text{OBS. 1. Schedulability is no worse using a fine-grained locking protocol than a similar coarse-grained one.} \)

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.

\( \text{OBS. 2. Resource replication improves schedulability.} \)

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

\( \text{OBS. 3. Fine-grained locking improves schedulability over coarse-grained locking most when the number of resources accessed within an outermost critical section is small.} \)

\( \text{\textsuperscript{7}Tighter analysis than that employed in these experiments is possible using an exponential-time algorithm. Such analysis, while perhaps computationally tractable for a single task system, is intractable when evaluating hundreds of thousands of task systems.} \)
This observation is corroborated by Fig. 10, which depicts the schedulability under the K-RNLP 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 affect 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 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


APPENDIX

In these appendices, we provide detailed proofs of claims in Secs. 5, and 4.

A. DGL ANALYSIS

Let \( w(\mathcal{R}_i, t) \) be the set of resources upon which \( \mathcal{R}_i \) is waiting at time \( t \). We say \( J_i \) is directly blocked by all jobs with earlier timestamps in any of the resource queues that \( J_i \) is enqueued in.

\[
DB(\mathcal{R}_i, t) = \{ \mathcal{R}_x \in \mathcal{R}_Q | \ell_x \in w(\mathcal{R}_i, t), t_s(\mathcal{R}_x) < t_s(\mathcal{R}_i) \}
\]

Additionally, it is possible that a job \( J_i \) is blocked because a request with an earlier timestamp could make a nested request for the resource for which \( J_i \) is waiting. We call this indirect blocking.

\[
IB(\mathcal{R}_i, t) = \{ \mathcal{R}_x \in \mathcal{R}_Q | \ell_x \in \mathcal{L} \wedge w(\mathcal{R}_i, t) \cap \mathcal{L}_x \neq \emptyset \wedge t_s(\mathcal{R}_x) < t_s(\mathcal{R}_i) \}
\]

The set of all jobs that \( J_i \) is blocked by at time \( t \) is the transitive closure of both direct and indirect blocking.

\[
B(\mathcal{R}_i, t) = \bigcup_{\mathcal{R}_x \in DB(\mathcal{R}_i, t)} DIB(\mathcal{R}_x, t)
\]

where \( DIB = DB(\mathcal{R}_i, t) \cup IB(\mathcal{R}_i, t) \). Note that \( DB(\mathcal{R}_i, t) \cup IB(\mathcal{R}_i, t) \subseteq B(\mathcal{R}_i, t) \).

From the definition of \( B(\mathcal{R}_i, t) \), similar to Lemma 1 of [12], we have the following.

\textbf{Lemma 1.} For any request \( \mathcal{R}_i \) and any time \( t \), \( \forall \mathcal{R}_x \in B(\mathcal{R}_i), t_s(\mathcal{R}_x, t) < t_s(\mathcal{R}_i, t) \).
To ensure a bounded duration of pi-blocking, we require that all jobs that are pi-blocked make progress. We thus require that Property P1 from [12] be upheld by any progress mechanism employed in an RSM supporting DGLs (Rules D1 and D2).

P1 If \( J_i \) is progress blocked (s-oblivious, s-aware, or spin-based) by the RSM, then \( J_i \) makes progress.

We then have the following, similar to Theorem 1 of [12].

**Theorem 1.** A job can be blocked by at most \( T - 1 \) (recall that \( T \) is the number of tokens in the RNLP) outermost requests within an RSM supporting DGLs (Rules D1 and D2).

**Proof.** By Property P1 a job that is pi-blocked makes progress. By Lemma 1, a request can never be blocked by another request with a later timestamp. Because there are at most \( T \) jobs with tokens, a job can be request blocked by at most \( (T - 1) \) requests with earlier timestamps. 

Lemma 1 and Theorem 1 parallel Lemma 1 and Theorem 1 of [12], and thus the duration of pi-blocking, irrespective of overheads, for DGLs is the same as requesting the set of resources in a nested fashion. This is because the set of requests that block \( R_j \) is the same under either policy due to the non-greedy nature of the resource queues.

**Progress mechanisms for DGLs.** Property P1 can be satisfied using priority boosting or priority donation under DGLs (see [12] for more detail). However, under the sporadic task model, which most existing schedulability tests assume, two jobs cannot inherit the priority of one job simultaneously. Thus, if priority inheritance is used to ensure progress, then we must be careful to ensure that this property is not violated.

The priority inheritance rule of the RNLP (Rule H1 of [12]), simply stated, allows only the earliest timestamp job that blocks a job \( J_i \) to inherit \( J_i \)’s priority. This inheritance rule was defined to allow a job that either transitively or indirectly blocks \( J_i \) to inherit \( J_i \)’s priority, and ensure progress.

However, there is a single job with the earliest timestamp that blocks \( J_i \) (because timestamps are totally ordered). Thus, a job’s priority can be inherited by at most one job at a time, and hence, this inheritance rule also supports DGLs.

Note that DGL requests and nested requests can both be supported by an RNLP instance. A job may make a request for a set of resources, execute a conditional statement, and then make a nested request for another set of resources. In hard real-time systems, however, it may be preferable to make a single request for all of the resources that may be needed in an outermost critical section to avoid excess context switches. This may be a rather conservative approach, particularly if most of the resources are not used. However, the conservative, non-greedy nature of Rule Q3 would have prevented other jobs from acquiring these resources under the original RNLP. Thus, the blocking term is no different, with the exception of the differences in context-switch overheads.

**B. 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 \( R_s \) is a long request, then we let \( N^s_{i,k} \) denote the number of nested short request within it.

Thus, a job’s priority can be inherited by at most one job at a time. Also, assume that DB(\( R_s \)) is the original definition of direct blocking from [12]. If \( ts(\mathcal{R}_i) < ts(\mathcal{R}_a, t) \) for some \( t \), then the new definition of direct blocking, denoted \( DB(\mathcal{R}_a) \), is equal to \( DB(\mathcal{R}_a) \) otherwise, \( DB(\mathcal{R}_a) = DB(\mathcal{R}_a) \). 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 long resource request, then we let \( DB(\mathcal{R}_a) \).

**Lemma 2.** A job is never blocked by a long critical section of a job with a later timestamp.

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

**Lemma 3.** Within the long outermost critical section of \( \mathcal{R}_a \), \( \mathcal{R}_a \) can be blocked by at most one outermost short request with a later timestamp for each short request nested within \( \mathcal{R}_a \).

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

**Lemma 4.** An outermost long request $R_i$ can be request blocked by the RSM for a total duration of at most $(T_iN_{\text{max}} + T_s)L_{\text{max}} + (T_i - 1)L_{\text{max}}$.

**Proof.** By Lemma 2, $R_i$ 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 $T_i - 1$ long request-holding requests with earlier timestamps than $R_i$, each of which executes for up to $L_{\text{max}}$ time. This accounts for $(T_i - 1)L_{\text{max}}$ blocking. It is also possible for $R_i$ to issue nested requests for short resources and be blocked by $T_s$ short requests with earlier timestamps, resulting in up to $T_sL_{\text{max}}$ units of blocking from short requests with earlier timestamps. Finally, by Lemma 3, $R_i$ and up to $T_i - 1$ other long requests with earlier timestamps that block $R_i$ may each be blocked by an outermost short request with a larger timestamp for each nested short request they make. This creates up to $T_iN_{\text{max}}L_{\text{max}}$ additional blocking due to short requests with later timestamps.

Thus, a job can hold the long token for at most $(T_iN_{\text{max}} + T_s)L_{\text{max}} + T_iL_{\text{max}}$ time.

**Lemma 5.** An outermost short request $R_i$ can be request blocked by the RSM for a total duration of at most $(\min(m, T)L_{\text{max}}(m - T_s) - 1)) + (\min(m, T) - 1)L_{\text{max}}$.

**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 $m - T_s$. The rest of the proof is similar to that of Theorem 2: we have to account for the token hold time of up to $m - T_s - 1$ requests and the request blocking experienced by $R_i$ while it holds the token. By Lemma 5, the former is at most $\max(0, \min(m, T)L_{\text{max}}(m - T_s) - 1))$ and the latter is at most $(\min(m, T) - 1)L_{\text{max}}$.

According to Theorem 3, it is likely best in practice to set $T_s = m$, for in this case, the total worst-case request blocking per outermost short request is merely $(m - 1)L_{\text{max}}$, which is independent of $T_i$.

**Theorem 3.** An outermost short request $R_i$ can be request blocked for a total duration of at most

$$\max(0, \min(m, T)L_{\text{max}}(m - T_s) - 1)) + (\min(m, T) - 1)L_{\text{max}}.$$