Nested 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 information about resource-usage patterns is known. These sources of blocking stem from system overheads 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 and resources to be replicated.

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 [10]. In recent work, we developed the first such protocol for multiprocessor real-time systems: the real-time nested locking protocol (RNLP) [10]. 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.

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 the system. In contrast, under coarse-grained locking, resource groups must be statically created before execution.

Motivation. After developing the RNLP, we attempted to apply it to manage shared caches on a multicore machine by treating cache lines as resources that tasks must acquire via a fine-grained locking protocol. To do so, we built upon the well-studied idea of cache coloring, and required that tasks lock all colors associated with their memory references before their execution. The goal of this work is to ensure that memory references of concurrently executing tasks on multiprocessors do not conflict in the cache, which should improve worst-case execution times, and make such execution times more amenable to timing analysis.

For set-associative caches, which are found in most commodity processors, we can model each way of associativity as a replica or unit of the color. Therefore, tasks may not need to lock the entire color, but may instead lock an arbitrary subset of the replicas of the color. Other tasks may then concurrently hold other replicas of the same color. To lock multiple replicas of multiple colors, a multi-unit multi-resource lock is required that allows tasks to lock one or more instances of multiple resources concurrently. The original RNLP is only a multi-resource lock, and therefore does not support such functionality.

In this cache-management application, a task may concurrently require a large number of resources, depending upon its working set size, i.e., how much cache space it requires. With traditional fine-grained locking, tasks must acquire all resources serially, which is overhead prone. If instead, tasks could atomically request a set of resources, the overhead associated with lock and unlock system calls could be drastically reduced.

We also applied the RNLP to a system with multiple graphics processing units (GPUs). Modern GPUs have independent engines to handle memory copies between GPUs or to system memory, as well as the execution of code on the GPU itself. By applying the RNLP to control access to these engines, we sought to improve performance by allowing for more fine-grained control of these shared resources. In this application, tasks may require exclusive access to one GPU execution engine, or two GPU copy engines to transfer data between GPUs.

Contributions. In this paper, we present a hybrid of coarse-and fine-grained locking in the context of the RNLP, which we call dynamic group locking, in which tasks atomically request a set of resources. For suspension-based locks, which require operating system support for both lock and unlock operations, dynamic group locks (DGLs) reduce the number of system calls (and therefore overhead) required over fine-grained locking. DGLs are particularly useful when the number of resources required is large, as in our cache-management application, though they are also valuable for tasks that require only two resources, for example, to copy data between shared objects, such as GPUs. In either case, the total number of lock and unlock calls is limited to one instead of the number of resources

∗Work supported by NSF grants CNS 1016954 and CNS 1115284; ARO grant W911NF-09-1-0535; AFOSR grant FA9550-09-1-0549; and AFRL grant FA8750-11-1-0033. The first author was also supported by an NSF fellowship.
requested. In this paper, we formalize DGLs, and show that they have the same worst-case blocking complexity as fine-grained locking under the RNLP while reducing the number of system calls.

We then build upon the RNLP by adding support for nested requests among replicated resources, which requires altering some of its queue structures to allow multiple tasks to hold replicas of the same resource concurrently. The resulting parallelism can then be reflected in the blocking analysis.

**Prior work.** Multiprocessor real-time locking protocols have been presented in several prior papers [5, 8, 9]. Of most relevance to this paper is work of Brandenburg and Anderson [3], who defined two definitions of priority-inversion blocking (pi-blocking) for suspension-based multiprocessor locking protocols (described more in the next section). They also developed the OMLP [3], and showed the OMLP and the FMLP [1] to be optimal for each of these definitions of pi-blocking. They also developed a $k$-exclusion\(^1\) variant of the OMLP that is optimal for clustered systems [4]. Ward et al. [11] also developed the R\(^2\)-DGPLP, a $k$-exclusion lock that is optimal for globally-scheduled system. However, to our knowledge, no existing multiprocessor locking protocols support either atomic requests of multiple resources, or tasks that request multiple instances of multi-unit resources.

**Organization.** In Secs. 2–3, we present background material and review the RNLP. In Sec. 4 we present DGLs and in Sec. 5 we add support for replicated resources. In Sec. 6, we present a brief schedulability study for our motivating application of shared caches. We conclude in Sec. 7.

## 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 $d_i$. We say that a released job is pending until it finishes its execution.

**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 $R_{i,k}$. 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 $\frac{n}{c}$ clusters of size $c$ each.

Each task has a base priority dependent upon the particular scheduling policy. A progress mechanism, which is employed by a locking protocol, can alter a job’s priority such that it has a higher effective priority. Three such 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 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.

**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 [3] 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). Under s-oblivious analysis, the suspensions of higher-priority work are analytically treated as computation time, where as under s-aware analysis, they are not. The choice of how suspensions are analyzed can drive the design of locking protocols as some protocols perform better under different types of analysis.

Under either definition of pi-blocking, tasks may be pi-blocked for two different reasons. A task may be pi-blocked while it waits to acquire shared resources, which we call request blocking. Additionally, tasks may be pi-blocked on account of a progress mechanism. We call this form of pi-blocking progress blocking. For example, if a high-priority job is released and is forced to donate its priority to a lower-priority resource-holding job it is said to be progress blocked.

**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, 3, 4, 10].

---

\(^1\)In a $k$-exclusion lock, tasks request one of $k$ units of a multi-unit resource.
### Analysis

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Scheduler</th>
<th>( k )</th>
<th>Progress Mechanism</th>
<th>Progress 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>( nL_{\text{max}} )</td>
<td>((n - 1)L_{\text{max}})</td>
</tr>
<tr>
<td></td>
<td>Clustered</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(^1)</td>
<td>( n )</td>
<td>Inheritance</td>
<td>( O(n) )</td>
<td>((n - 1)L_{\text{max}})</td>
</tr>
</tbody>
</table>

\(^1\) Applicable only under certain schedulers such as EDF and rate monotonic.

| s-oblivious | Partitioned | \( m \) | Donation | \( mL_{\text{max}} \) | \((m - 1)L_{\text{max}}\) |
|            | Clustered | \( m \) | Donation | \( mL_{\text{max}} \) | \((m - 1)L_{\text{max}}\) |
|            | Global | \( m \) | Donation | \( mL_{\text{max}} \) | \((m - 1)L_{\text{max}}\) |

\( L_{\text{max}} \) 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.

### 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 requests. The overall architecture of the RNLP is shown in Fig. 2. 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 progress mechanisms can be combined to form both efficient spin- and suspension-based locking protocols.

The token lock is effectively a \( k \)-exclusion lock that serves to limit the number of jobs that can have complete resource requests at a time. Therefore, existing \( k \)-exclusion locks can be employed as token locks [4, 11].

The RSM controls access to all shared resources. 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 job at the head of \( \text{RQ}_a \) from acquiring \( \ell_a \) if another job 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 original rules of the RSM are given below.\(^2\) In these rules, \( L_{i,k} \) denotes the set of resources that \( J_i \) may request in its \( k \)-th outermost critical section (including nested requests).

---

\(^2\) We adapted the rules in [10] to be in terms of requests instead of jobs for notational simplicity later. The two sets of rules are functionally identical.

---

1. **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: \( ts(\text{R}_{i,k}) := t \). We assume a total order on such timestamps.
2. **Q2** All jobs with requests in \( \text{RQ}_a \) wait with the possible exception of the job whose request is at the head of \( \text{RQ}_a \).
3. **Q3** A job \( J_i \) with an incomplete request \( \text{R}_{i,k} \) acquires \( \ell_a \) when it is the head of \( \text{RQ}_a \), and there is no request \( \text{R}_{x,y} \) with \( ts(\text{R}_{x,y}) < ts(\text{R}_{i,k}) \) such that \( \ell_a \in \text{L}_{x,y} \).
4. **Q4** When a job \( J_i \) issues a request \( \text{R}_{i,k} \) for resource \( \ell_a \), \( \text{R}_{i,k} \) is enqued in \( \text{RQ}_a \) in increasing timestamp order.
5. **Q5** When a job releases resource \( \ell_a \) it is dequeued from \( \text{RQ}_a \) and the new head of \( \text{RQ}_a \) can gain access to \( \ell_a \), subject to Rule Q3.
6. **Q6** When \( J_i \) completes its outermost critical section, it releases its token.

**Example.** To illustrate the key concepts of the RNLP, consider a globally-scheduled EDF system with \( m = 2 \) processors, \( T = 2 \) tokens, and \( q = 2 \) resources, \( \ell_a \) and \( \ell_b \), as seen in Fig. 3. 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 \( ts(\text{R}_{1,1}) = 1 \), and then immediately acquires \( \ell_a \). At time \( t = 2 \), \( J_2 \) requires \( \ell_b \), and it acquires a token with timestamp \( ts(\text{R}_{2,2}) = 2 \). However, because \( J_1 \) could request \( \ell_b \) 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 \( \ell_a \) 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 \( t = 8 \) when \( J_3 \) releases \( \ell_b \). Similarly, we assume that the acquisition of a token and subsequent enqueuing into the associated RQ occur atomically.

---

This rule was presented as Rule M1 in the online appendix of [10], though it generalizes the original Rule Q3.

---

\[^{1}\] This rule was presented as Rule M1 in the online appendix of [10], though it generalizes the original Rule Q3.

\[^{2}\] We adapted the rules in [10] to be in terms of requests instead of jobs for notational simplicity later. The two sets of rules are functionally identical.
at time $t = 4$, $J_4$ requires $\ell_a$ but there is not an available token. $J_4$ suspends until $t = 8$ when $J_2$ finishes its outermost critical section and releases its token. However, at $t = 8$, $\ell_a$ 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 pi-blocking bounds [10]. In this table, bounds on progress blocking and request blocking (see Sec. 1) are listed separately.

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, while reducing system call overheads.

Rules. To enable the use of DGLs in the RNLP, we modify the protocol as follows. When a job $J_i$ issues a request for a set of resources $R$, 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 $\{RQ_a | \ell_a \in R\}$. The DGL request is satisfied when it has acquired all resources in $R$, 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 $R_{i,k}$ for a subset of resources $R \subseteq L$ acquires all resources in $R$ when $R_{i,k}$ is the head of every resource queue associated with a resource in $R$, and there is no request $R_{x,y}$ with $ts(R_{x,y}) < ts(R_{i,k})$ for which there exists a resource $\ell_a \in R \cap L_{x,y}$.

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

In Appendix A, we prove that the RNLP with DGLs has the same worst-case blocking bounds as the original. Intuitively, such bounds do not change because a DGL request enqueues in a single queue and essentially “reserving” slots in other queues (Rule Q3) for potential future nested requests. Thus, in the worst case, the set of blocking requests is the same.

If all concurrent resource accesses in a system are supported by 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 because 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 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 [6, 7, 11]. 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 [6], the $k$-FMLP [7], and the $R^2$DGLP [11], which function as depicted in Fig. 4. In these protocols, each replica is
and each request accesses replicas of different resources. This rule causes problems with replicated resources. Consider again Fig. 5. Consider an outermost request \( R_{i,k} \) for Resource A that may make a nested request for Resource B. Which replica queue for Resource B should hold its “reservation”? If a specific queue is chosen by the “shortest queue” rule when \( R_{i,k} \) receives its timestamp, and if \( R_{i,k} \) does indeed generate a nested request for Resource B later, then the earlier-selected queue may not still be the shortest for Resource B when the nested request is made. If a queue is not chosen until the nested request is made, then since \( R_{i,k} \) had no “reservation” in any queue of Resource B until then, it could be the case that requests with later timestamps hold all replicas of Resource B when the nested request is made. This violates a key invariant of the RNLP.

Our solution is to require \( R_{i,k} \) 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 \( R_{i,k} \) when it receives a token that enqueues a “place-holder” request for \( R_{i,k} \) 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.

6 Results

We demonstrate the efficacy of the presented techniques within the framework of our proposed cache management application. By requiring tasks to lock regions of the cache that they will use, we guarantee cache isolation among tasks on different processing cores. In our experimental results on real hardware, we observed worst-case execution times (WCETs) decreased to \( \frac{1}{2} \) or \( \frac{1}{4} \) of their execution cost as compared to when tasks were allowed free access to the cache. However, these decreased WCETs come at the cost of increased blocking while waiting for regions of the cache to become available. In our evaluations we evaluate this tradeoff between improved WCETs and increased blocking, and show how the techniques presented in this paper improve overall system schedulability.

To reflect the real system on which we implemented our cache management scheme, we assume a system with four processors and a 16-way set associative 8MB cache. We used partitioned rate monotonic scheduling, and assigned tasks to processors using the worst-fit heuristic. Due to space constraints, we will describe the experimental setup for the results shown in Fig. 6, though we tested many other scenarios. All tasks had a cache footprint of 2560KB, such that the cache was not partitionable, and colors were assigned to tasks in blocks of four ways at a time so as to minimize the total number of ways required of any color. Tasks were assumed to have uniformly distributed utilizations in the range of \([10,100]\)ms, with harmonic periods in \([25,50,100,200]\)ms. We applied job and period slicing techniques such that individual subjobs had periods no greater than 25ms, and subjobs had execution times no more than 2.5ms, which improved our worst-case blocking bounds.

In Fig. 6 the solid (dashed) lines depict the schedulability when cache lines are locked (not locked). To account for the greater WCETs when cache lines are not locked, we multiplicatively scale all execution time by a factor \( S \in \{2, 3, 4, 8\} \). From these results, we can draw a few conclusions. First, the proposed RNLP extensions, denoted RNLP-E, greatly improve schedulability over the original RNLP. Second, our proposed cache management scheme has the same schedulability as a task sys-

\[\text{Size of all cache lines that the task may ever use.}\]
system without our scheme but with double the execution time, and much better schedulability when compared to higher scaling factors commensurate with our experimental observations. Also note, while our proposed cache management scheme does not schedule more than a utilization of two on a four core machine, a utilization of two under our scheme corresponds with a utilization of eight or more when cache locking is not applied, and such task systems are clearly infeasible without our scheme.

7 Conclusions

We have presented two extension of the RNLP [10] that address issues of practical concern that arise when attempting to support nested resource requests in a real-time multiprocessor system in a fine-grained way. First, we introduced dynamic group locks (DGLs) to reduce system call overhead by allowing tasks to atomically request a set of resources with a single system call. The RNLP with DGLs therefore generalizes standard group locking, while also allowing for such locks to be nested. We also showed how to incorporate replicated resources within the RNLP, which has analytical benefits when tasks only require some replica of a resource, instead of a particular one. Finally, we conducted a brief schedulability study to demonstrate the utility of the proposed extension of the RNLP.

References


A DGL Analysis

Let \( w(R_{i,k}, t) \) be the set of resources upon which \( R_{i,k} \) 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.

\[
DB(R_{i,k}, t) = \{ R_{x,y} \in RQ_a | \ell_a \in w(R_{i,k}, t), \\
\quad ts(R_{x,y}) < ts(R_{i,k}) \}
\]

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

\[
IB(R_{i,k}, t) = \{ R_{x,y} \in RQ_a | \ell_a \in \mathcal{L} \land \\
\quad w(R_{i,k}, t) \cap \mathcal{L}_{x,y} \neq \emptyset \land \\
\quad ts(R_{x,y}) < ts(R_{i,k}) \}
\]

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

\[
B(R_{i,k}, t) = \bigcup_{R_{x,y} \in DB(R_{i,k}, t)} DIB(R_{x,y}, t)
\]

where \( DIB = DB(R_{i,k}, t) \cup IB(R_{i,k}, t) \). Note that \( DB(R_{i,k}, t) \cup IB(R_{i,k}, t) \subseteq B(R_{i,k}, t) \).

From the definition of \( B(R_{i,k}, t) \), similar to Lemma 1 of [10], we have the following.

**Lemma 1.** For any request \( R_{i,k} \) and any time \( t \), \( \forall R_{x,y} \in B(R_{i,k}), ts(R_{x,y}) < ts(R_{i,k}) \).

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 [10] 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.

This property can be satisfied, depending upon the system, using priority boosting, priority donation, or priority inheritance (n.b., priority inheritance must be transitive). With such a progress mechanism in place, we then have the following, similar to Theorem 1 of [10].

**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 [10], 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 \( R_{i,k} \) is the same under either policy due to the non-greedy nature of the resource queues.