Replica-Request Priority Donation: A Real-Time Progress Mechanism for Global Locking Protocols *

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

Abstract

Real-time locking protocols employ progress mechanism(s) to ensure that resource-holding jobs are scheduled. These mechanisms are required to bound the duration of priority-inversion blocking (pi-blocking) for jobs sharing resources. Examples of such progress mechanisms include priority inheritance and priority donation. Unfortunately, some progress mechanisms can cause any job, including those that never request shared resources, to be blocked upon job release. This paper presents a variant of priority donation for globally-scheduled systems that only causes blocking for jobs waiting for shared resources. Additionally, this variant of priority donation is employed to construct a new suspension-based locking protocol called the replica-request donation global locking protocol (R^2DGLP), which is optimal for both mutex and k-exclusion (i.e., multi-resource) locks. This work is motivated by multicore systems where tasks may share I/O devices (e.g., GPUs) where critical sections can be long. In such applications, progress mechanisms that cause jobs that do not access I/O devices to be blocked to ensure progress can be detrimental from a schedulability perspective.

1 Introduction

When resources are shared among tasks in a multiprocessor real-time system, multiprocessor synchronization algorithms must be coupled with scheduling algorithms to ensure that blocking caused by synchronization does not cause timing violations. To bound the duration of time a job can be blocked, real-time locking protocols employ *progress mechanisms* that ensure that resource-holding jobs are scheduled, and therefore that blocked jobs make *progress* towards acquiring the shared resources they require. These progress mechanisms are dependent upon the scheduler employed.

A wide range of such schedulers can be viewed as instantiations of *clustered* scheduling, wherein processors are partitioned into clusters (often representative of the underlying hardware architecture), with each task assigned to a single cluster. Within each cluster, tasks are scheduled from a single ready queue and may migrate among processors within the cluster. Note that clustered scheduling generalizes both *partitioned* and *global* scheduling in which, respectively, each processor is a cluster, or all processors form a single cluster. Recent experimental work [2, 4] has demonstrated that partitioned scheduling is often preferable for hard-real-time systems (i.e., systems in which deadlines cannot be missed), while clustered scheduling is often preferable for soft-real-time system (i.e., systems in which deadlines can be missed by a bounded duration of time) [7, 16].

When resources are shared among clusters, the mechanisms required to ensure progress can cause blocking for all jobs in the system upon job release, not just those that engage in the locking protocol [6]. Intuitively, this is caused when the execution of a higher-priority job must be delayed upon its release so that a lower-priority resource-holding job in the same cluster is not preempted, thereby ensuring progress for resource-requesting jobs in other clusters. We call this type of blocking *release-blocking*, in contrast to the blocking experienced by jobs waiting for shared resources, which we call *request-blocking*. Examples of progress mechanisms that induce release-blocking include non-preemptive sections, and *job-release priority donation* (*JRPD*) [6], which will be described more formally later. An example schedule in which both release- and request-blocking occur is given in Fig. 1. Both optimal and heuristic algorithms have been developed to pack tasks onto clusters to reduce sharing across clusters and therefore minimize the effect of release-blocking [14, 15]. However, this problem has been shown to be $\mathcal{NP}$-hard in the strong sense [15].

Locking protocols developed for sharing resources between clusters can still be used when resource sharing is strictly local within a cluster. However, the progress mechanisms that induce release-blocking, necessary for inter-cluster resource sharing, are no longer necessary—alternative progress mechanisms that induce only request-blocking are therefore often favorable. For example, it has been shown that locking protocols can be employed in real-time applications where graphics processing units (GPUs) perform general-purpose computations. These locking protocols enable guarantees on real-time constraints to be made since they subsume the role of resource arbitration from non-real-time GPU driver software [9, 10, 11]. However, the critical sections associated with general-purpose com-

*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.

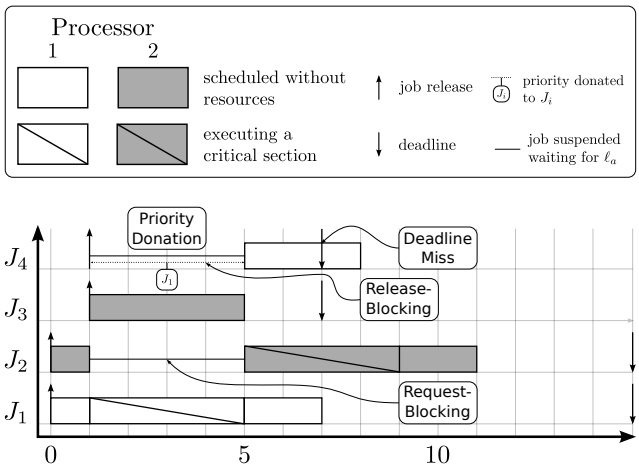

Figure 1: Example of the effects of release-blocking under JRPD on an earliest-deadline-first globally-scheduled system with $m = 2$ processors. J_4 is release-blocked at time $t = 1$ while it donates its priority to J_1. This release-blocking causes J_4 to miss its deadline. If instead, J_4 had preempted J_1, J_1 and J_2 would finish three time units later, but still meet their deadlines. Also, at time $t = 1$, J_2 is request-blocked by J_1, which holds the shared resource.

putation on GPUs can be very long: typically tens of milliseconds, but even as great as several seconds, in length. These lengths are orders of magnitude greater than those normally associated with shared data structures [4], a more typical application of locking protocols.

The use of locking protocols that induce release-blocking for resources shared strictly locally within a cluster may be acceptable (though inefficient) with respect to schedulability, when critical section lengths are short. However, schedulability can easily become infeasible when critical section lengths are long. For example, it is impossible to schedule a job that has a response-time constraint shorter than the longest critical section, since such a job can be release-blocked for the duration of the longest critical section. Therefore, in a typical GPU case, no task may have a response-time requirement less than tens of milliseconds. This can significantly limit the flexibility of a GPU-enabled real-time system in that tasks with short response-time constraints cannot be scheduled within the same cluster as GPU tasks. However, if a locking protocol that only uses progress mechanisms that incur request-blocking is used, then a GPU-enabled real-time system can schedule tasks with a wide array of response-time constraints, scheduling tasks with both short and long response-time requirements.

The blocking behavior of locking protocols is analyzed on the basis of *priority inversion blocking* (*pi-blocking*), or the duration of time that a job is blocked while a lower-priority job is executing. The duration of pi-blocking is then treated as CPU demand (execution time) under most schedulability analysis techniques for globally-scheduled multiprocessor systems, thereby increasing system utilization. This motivates another reason why release-blocking should be avoided whenever possible: generally speaking, workloads with greater utilizations are more difficult to schedule. Release-blocking affects every task and can thus have a cumulative affect on the system utilization, particularly when there are many tasks in the system. This is especially unfortunate when only a few tasks actually access shared resources, since this may still result in significant system-wide effects. The situation is even worse when per-task release-blocking is large, such as in the case of GPUs.

Motivated by applications with long critical sections, such as systems containing GPUs, and the desire to eliminate release-blocking, we focus on globally scheduled systems, though our results can be applied locally within a cluster. We assume a *job-level fixed priority* (*JLFP*) system, in which a each job's priority is constant (e.g., *earliest deadline first* (*EDF*) or *static priority scheduling*). We also assume that jobs *suspend* while waiting for shared resources instead of *busy-waiting* or *spinning*. By suspending, the processor is made available for other tasks to execute, which can improve response times, particularly when critical sections are long as is the case with GPUs.

Contributions. We have developed *replica-request priority donation* (*RRPD*), a progress mechanism, which is a variant of JRPD [6], for globally scheduled systems and within a local cluster, that does not cause release-blocking. We then construct the R^{2}DGLP,[1] an optimal k-exclusion[2] suspension-based locking protocol based on RRPD and priority inheritance. We compare the R^{2}DGLP with existing locking protocols and demonstrate the performance improvements.

Prior work. Many suspension-based multiprocessor real-time locking protocols such as the MPCP [17], DPCP [17], PPCP [8] and the MSRP [13] are multiprocessor extensions of their uniprocessor counterparts, the stack resource policy (SRP) [1] and the priority ceiling protocol (PCP) [1, 17]. Recent research has built upon these results through the development of new multiprocessor locking protocols that are asymptotically optimal, which we discuss next.

Brandenburg and Anderson [5] developed two definitions of priority-inversion blocking: *suspension-oblivious* (*s-oblivious*) and *suspension-aware* (*s-aware*). Under s-oblivious analysis, the suspensions of higher-priority jobs are considered computation time analytically, whereas under s-aware analysis, these suspension are modeled analytically. They also established per-request lower bounds of $\Omega(m)$ and $\Omega(n)$ for s-oblivious pi-blocking and s-aware pi-blocking, respectively, on any JLFP system (partitioned, clustered or global), for mutual exclusion (mutex) resources where m is the number of processors and n is the number of tasks in the task system.

Block et al. [3] developed the FMLP, which was later proven to be asymptotically optimal under s-aware anal-

[1] Previously called the I-KGLP in [18], and described in an unpublished appendix available online only.

[2] A k-exclusion lock is a generalization of mutex lock in which there are k identical serially reusable replicas.

ysis. The FMLP uses non-preemptive sections to ensure progress. Later, Brandenburg [4] extended the FMLP to the *FIFO mutex locking protocol* (*FMLP*$^+$), which is more flexible because resource-holding jobs can execute either preemptively or non-preemptively. The FMLP$^+$ is optimal under s-aware analysis for a large class of global schedulers, including global EDF (G-EDF), for task systems with job deadlines equal to their periods.[3] Brandenburg and Anderson also developed the $O(m)$ *locking protocol* (*OMLP*) family of locking protocols, which are optimal under partitioned-, clustered-, and globally-scheduled systems under s-oblivious analysis for mutex resources [5, 6].

The clustered variant of the OMLP (C-OMLP) employs a progress mechanisms called priority donation, which, as noted earlier, we refer to as job-release priority donation (JRPD). In JRPD, a job J_d that would cause a preemption of a resource-holding job J_i suspends and donates its priority to J_i [6]. J_d is then said to be the *priority donor* of J_i. A donation relationship is static and persists until J_i completes its critical section, or another higher-priority job relieves J_d of its donation obligation. This property ensures that a resource-holding job is always scheduled. Priority donation is very general and has been used to construct optimal mutex, k-exclusion, and reader-writer locking protocols [6]. However, priority donation can also cause release-blocking for all jobs in the system, in addition to the request-blocking incurred for each request. An example of JRPD, and the consequences of long release-blocking are depicted in Fig. 1.

In work on k-exclusion locking protocols, Elliott and Anderson extended the FMLP to the k-FMLP, which is optimal under s-aware analysis [11] for the same class of global schedulers for which the FMLP$^+$ is optimal. Elliott and Anderson also developed the O-KGLP which is asymptotically optimal under s-oblivious analysis [9]. The O-KGLP is composed of a priority queue, and a FIFO queue per resource. It also employs a variant of priority donation that is initiated upon request, similar to the progress mechanism we present. However, their definition of priority donation can cause additional pi-blocking, which we eliminate in this work.

Organization. In Sec. 2, we formally define our system model and specify our assumptions. In Sec. 3, we describe a new variant of priority donation called replica-request priority donation (RRPD). In Sec. 4, we present the R^{2}DGLP, a k-exclusion locking protocol based upon RRPD. In Sec. 5, we compare the R^{2}DGLP to previous suspension-based locking protocols through schedulability experiments reflective of systems with GPU-inspired schedulability constraints. Finally, we conclude in Sec. 6.

[3] S-aware optimality under clustered systems, including globally-scheduled systems, is still an open issue.

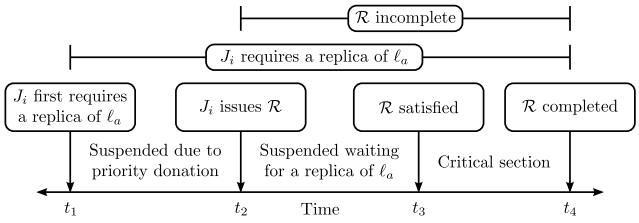

Figure 2: Phases of a replica acquisition under RRPD.

2 Background and Definitions

We consider a system of n sporadic tasks $\{T_1, \ldots, T_n\}$ scheduled on m processors. Each task is composed of a sequence of jobs; we let $J_{i,k}$ denote the k^{th} job of the i^{th} task, though we omit the job index when it is insignificant. A task is characterized by its *worst-case execution time*, e_i, *minimum job separation*, p_i, and its *relative deadline*, d_i. Each job is said to be *released* when it is available for execution and *pending* until it finishes its execution. A pending job can be either *ready* or *suspended*. A job is said to be ready if it is available for execution, whereas a suspended job cannot be scheduled.

Resources. The system also contains q shared resources, $\mathcal{L} = \{\ell_1, \ldots, \ell_q\}$ such as shared memory objects or I/O devices. Each resource ℓ_a may be a multi-unit resource, with k_a serially reusable units. Therefore, at most k_a requests for ℓ_a may be satisfied at a time, one for each replica. We assume that a job can only request one replica at a time. Note that in the case that $k_a = 1$, the resource is an ordinary serially reusable (mutex) resource.

Access to each resource is controlled by a *locking protocol*. Ordinarily, when a job J_i requires a replica of resource ℓ_a, it *issues* a *replica request* $\mathcal{R}$ to the locking protocol for a replica of ℓ_a. However, in the locking protocol we develop, the issuance of $\mathcal{R}$ can be deferred from the first instant at which J_i requires a replica of ℓ_a. A request is said to be *satisfied* when J_i acquires a replica of ℓ_a and *completed* when J_i releases its replica of ℓ_a. A job J_i is said to have an incomplete request during the interval of time between when the request is issued to when the request is completed. J_i is said to require a replica of ℓ_a from the first instant in time at which it requires a replica of ℓ_a through the time at which it completes its request. The segment of a job's execution between a request being satisfied and completed is called a *critical section*. These phases of replica acquisition are shown in Fig. 2.

We assume that replica requests are *non-nested*, i.e., a job holding a replica of ℓ_a cannot make another replica request for either a replica of ℓ_a or another resource ℓ_b until its current critical section is completed. Note, however, that coarse-grained nesting can be supported by *grouping* replicas and treating such a group as a single resource as in the FMLP [3].

Scheduling. We consider globally-scheduled JLFP systems, in which jobs are scheduled from a single ready

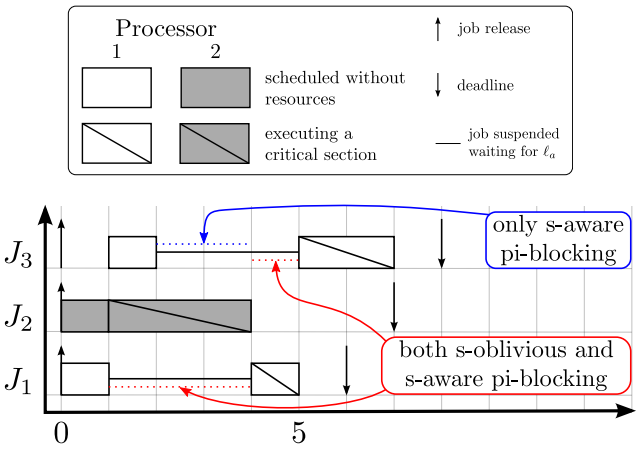

Figure 3: Illustration adapted from [5] of the difference between s-oblivious and s-aware analysis. In this example, three G-EDF-scheduled jobs share a single resource ℓ_a on two processors. During $[2, 4)$, J_3 is blocked, but there are m jobs with higher priority, thus J_3 is not s-oblivious pi-blocked. 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.

queue. Jobs are therefore free to migrate among all processors. Under a JLFP scheduler, each job has a fixed *base priority* that is assigned upon job release. However, progress mechanisms such as priority inheritance can alter a job's priority during its execution. A job's modified priority is called its *effective priority*. For example, under priority inheritance, a job holding ℓ_a has an effective priority equal to the highest base priority of any job waiting for ℓ_a. At any time, the m ready jobs with the highest effective priorities are scheduled.

Blocking. Brandenburg and Anderson [5] defined s-oblivious and s-aware blocking as follows.

Definition 1. Under **s-oblivious** schedulability analysis, a job J_i incurs *s-oblivious pi-blocking* at time t if J_i is pending but not scheduled and fewer than m higher-priority jobs are **pending**.

Definition 2. Under **s-aware** schedulability analysis, a job J_i incurs *s-aware pi-blocking* at time t if J_i is pending but not scheduled and fewer than m higher-priority jobs are **ready**.

An example of the difference between s-oblivious and s-aware pi-blocking is given in Fig. 3. In this paper we focus on s-oblivious analysis because it is used in almost all global schedulability analysis. As such, unless otherwise noted, when we say a job is release- or request-blocked, we assume it to also be s-oblivious pi-blocked.

Assumptions. In our analysis of worst-case pi-blocking bounds, we consider a number of variables such as the critical section length L^{max}, the frequency of replica requests, and number of requests each task T_i may issue N_i, to be constant. Fine-grained analysis of pi-blocking, which incorporates the specific values of many of these variables (e.g., per-request critical section lengths), can be conducted to achieve tighter pi-blocking bounds, using similar analysis techniques to that presented in [5, 4]. We assume that n and m are variable and all other parameters are constant, similar to the analysis assumptions of [5, 6, 18].

3 Replica-Request Priority Donation

Job-release priority donation (JRPD) was designed as a progress mechanism for clustered systems [6]. In a clustered-scheduled system, comparing priorities across clusters is not very useful, because a high-priority job with respect to one cluster may have a relatively low priority when compared to jobs in another cluster. To ensure progress, JRPD ensures that all resource-holding jobs are scheduled by forcing high-priority jobs to suspend and donate their priority upon release to prevent preemptions of resource-holding jobs. However, this donation can cause release-blocking for any job in the system, not just those that engage in the locking protocol. We demonstrate such behavior in Fig. 1, in which, at time $t = 1$, J_4 is released, and is forced to suspend and donate its priority to J_1, so that J_1 remains scheduled.

In a globally-scheduled system, priorities can be compared among all tasks, which allows us to adapt the rules of priority donation such that jobs that do not ever require shared resources are never pi-blocked. Thus, there is no release-blocking. To do so, we modify priority donation such that a job donates its priority on replica request instead of job release. We call this new definition of priority donation, *replica-request priority donation* (*RRPD*). Note that RRPD on its own is not sufficient to ensure progress, but when coupled with a progress mechanism such as priority inheritance, as we will demonstrate with respect to the R^{2}DGLP in Sec. 4, yields desirable worst-case pi-blocking bounds. The rules of the RRPD will be demonstrated later in an example of the R^{2}DGLP.

In the following rules, let J_i be a job that requires a replica of ℓ_a at time t_1. Let t_2 be the time that J_i issues its replica request. Let t_3 and t_4 be the times that J_i's request is satisfied and completed, respectively. These times are depicted in Fig. 2. Additionally, let J_d be a priority donor of J_i that requires a replica of ℓ_a at time t_x. J_d suspends to let J_i complete its replica request.

D1 J_i may issue a request for a replica of ℓ_a only if it is among the m jobs of highest effective priority that currently require a replica of ℓ_a (including jobs with an incomplete request for a replica of ℓ_a). If necessary, J_i suspends until it may issue its replica request.

D2 J_d becomes J_i's priority donor a time t_x if (a) J_d has one of the m highest base priorities among jobs that currently require a replica of ℓ_a, (b) J_i is the lowest effective-priority job[4] with an incomplete request for a

[4]Ties are broken according to the scheduler's tie-breaking rules.

replica of ℓ_a at time t_x, and (c) there are m jobs with an incomplete request for a replica of ℓ_a.

D3 J_i assumes the priority of J_d (if any) during $[t_2, t_4)$. J_d is considered to have no effective priority while it is a donor.

D4 If a job J_d donating its priority to J_i is displaced from the set of the m highest base-priority jobs that require a replica of ℓ_a by a job J_h, then J_h becomes J_i's priority donor and J_d ceases to be a priority donor. (By Rule D3, J_i thus assumes J_h's priority.)

D5 A priority donor is suspended throughout the duration of its donation.

D6 J_d ceases to be a priority donor as soon as either (a) J_i completes its critical section (i.e., at time t_4), or (b) J_d is relieved by Rule D4.

Note that Rules D1-D6 on their own do not necessarily ensure progress in their own right. For example, consider a system with two processors, and one replica of ℓ_a, in which two lower-priority jobs J_{l_1} and J_{l_2} have incomplete requests for a replica of ℓ_a, and two higher-priority jobs J_{h_1} and J_{h_2} are the priority donors of J_{l_1} and J_{l_2}, respectively. If J_i is released and has one of the highest m effective priorities in the system, then only one of J_{h_1} and J_{h_2} has sufficient priority to be scheduled. If J_{h_1} has a higher priority than J_{h_2}, but J_{h_1} is not donating to the replica holder, than progress is not ensured. For this reason, we require the rules of a locking protocol that utilizes RRPD to also ensure the following progress property.

P1 A job J_i with an incomplete replica request makes progress (i.e., the replica-holding job for which J_i is waiting is scheduled) if J_i has sufficient effective priority to be scheduled.

In the R^{2}DGLP, this property is satisfied through the use of priority inheritance.

Analysis. We next analyze several qualities of RRPD that will later be used to bound the duration of pi-blocking for the R^{2}DGLP.

Lemma 1. *A job J_i with an incomplete request for a replica of ℓ_a has one of the m highest effective priorities among jobs that require a replica of ℓ_a (those that have issued a request, as well as those that are suspended waiting to issue a request, as seen in Fig. 2).*

Proof. By contradiction. Assume J_i has an incomplete request for a replica of ℓ_a but does not have one of the m highest effective priorities among the jobs that require a replica of ℓ_a. Thus there are at least m jobs of higher effective priority than J_i that require a replica of ℓ_a. Then either all m of these higher effective-priority jobs have issued requests, or there is at least one such higher effective-priority job that is suspended waiting to issue its request by either Rule D1 or D5. We consider these cases separately.

If all m jobs with higher effective priority than J_i that currently require a replica of ℓ_a issued their replica requests before J_i, then J_i would not have been allowed to issue its request by Rule D1.

Consider a job J_d that is one of the m jobs of higher effective priority than J_i, which is suspended and waiting to issue its request for a replica of ℓ_a. J_d is not suspended by Rule D1, because it has one of the highest m effective priorities among jobs that currently require a replica of ℓ_a. Thus J_d must be suspended by Rule D2, and is thus a priority donor. Therefore J_d has no effective priority by Rule D3, because its priority is being donated to a job with an incomplete request for ℓ_a (e.g., J_i). □

Lemma 2. *There are at most m jobs with an incomplete request for a replica of ℓ_a at any time.*

Proof. Assume for contradiction that there are more than m jobs with an incomplete request for a replica of ℓ_a. Let J_i be the job that issued the $(m+1)^{st}$ request for a replica of ℓ_a. By Rule D1, when J_i issued its request, it was one of the m jobs of highest effective priority with an incomplete request for a replica of ℓ_a. However, there were also m jobs with incomplete requests for a replica of ℓ_a, that, by Lemma 1, had the highest m effective priorities of jobs that required a replica of ℓ_a. This contradicts the assumption that J_i was allowed to issue a replica request. □

Lemma 3. *A job J_i that has one of the highest m base priorities among jobs that currently require a replica of ℓ_a also has one of the highest m effective priorities (with respect to priority donation only) among jobs that currently require a replica of ℓ_a.*

Proof. The only way for a job's effective priority to be increased is through priority donation, or Rule D2. Priority donation forms a one-to-one relationship between donor and recipient, in which the recipient's effective priority is elevated while the donor's effective priority is reduced to zero. Thus, the highest m effective priorities are equal to the highest m base priorities among jobs that currently require a replica of ℓ_a. Therefore, if a job has one of the highest m base priorities, it is not a priority donor or recipient, and thus also has one of the highest m effective priorities (with respect to priority donation only) among jobs that currently require a replica of ℓ_a. □

Lemma 4. *Under RRPD, if a job J_i that requires a replica of ℓ_a is pi-blocked waiting for a replica of ℓ_a it either has an incomplete request for a replica of ℓ_a or it is a priority donor.*

Proof. Assume for contradiction that a job J_i is pi-blocked, does not have an incomplete request for a replica of ℓ_a, and is not a priority donor. By the definition of pi-blocking, if J_i is pi-blocked, then it has one of the highest m base priorities in the system, and by Lemma 3 is among the set of the highest m effective-priority jobs that need a replica of ℓ_a. Thus, by Rule D1, J_i would issue a request for a replica of

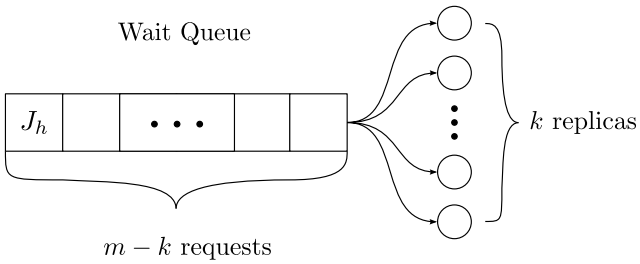

Figure 4: Queue structure used by the CK-OMLP, which is suboptimal when used with RRPD. Under JRPD, the progress mechanism employed by the CK-OMLP, all replica-holding jobs are scheduled, and thus the total pi-blocking is at most $\lceil (m-k)/k \rceil L^{max}$. However, under RRPD, if J_h is the only job with sufficient effective priority to be scheduled, then only one replica-holding job will be scheduled. Thus J_h must wait in the wait queue for $(m-k-1)L^{max}$, which is suboptimal.

ℓ_a. □

Lemma 5. *A priority donor J_d can be pi-blocked during priority donation for at most the maximum duration of time that a job can be pi-blocked with an incomplete request for a replica of ℓ_a (refer to timeline in Fig. 2), plus one critical section.*

Proof. If J_d is pi-blocked while it is a priority donor, then the recipient of its priority donation J_i has sufficient effective priority to be scheduled. By Property P1, J_i makes progress. Thus J_d can be pi-blocked while it is donor for at most the maximum duration of time J_i can be pi-blocked waiting for its request to be satisfied, plus J_i's critical section length. □

The rules of RRPD facilitate the design of a simple yet optimal k-exclusion locking protocol, which we present next.

4 R^{2}DGLP

The clustered k-exclusion OMLP (CK-OMLP) [6], which employs JRPD, uses a single FIFO-ordered queue to order the acquisition of replicas. Under JRPD, every replica-holding job is scheduled, and thus all requests make progress and the maximum duration of pi-blocking is $O(m/k)$. This design does not extend to RRPD. Under RRPD, a replica-holding job is not guaranteed to be scheduled (if higher-priority work is present), and thus if only one job J_i with an incomplete replica request has sufficient priority to be scheduled, only one replica-holding job would be scheduled. Thus it is possible for all requests to be serialized on a single resource, which results in an $(m-k-1)L^{max}$ blocking bound, which is suboptimal, as shown in Fig. 4. Instead the R^{2}DGLP employs a similar queue structure to the O-KGLP [9] and the k-FMLP [11], in which there are k_a queues for each resource ℓ_a, one per replica.

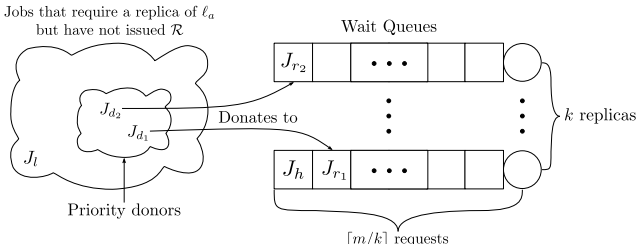

Figure 5: Queue structure of the R^{2}DGLP.

Structure. In the R^{2}DGLP, access to each replica of a resource ℓ_a is arbitrated by an individual FIFO ordered replica queue denoted KQ_x. Within each replica queue, priority inheritance is used to ensure progress. As will be proven later, this design limits the maximum queue length to $\lceil m/k \rceil$, and thus the maximum duration of pi-blocking is $O(m/k)$, which is optimal. The queue structure of the R^{2}DGLP is shown in Fig. 5. In the following rules and analysis, we consider, without loss of generality, only a single resource ℓ_a, with k replicas.

K1 J_i is enqueued on the shortest KQ_x when it issues $\mathcal{R}$. J_i suspends until $\mathcal{R}$ is satisfied (if KQ_x was non-empty).

K2 $\mathcal{R}$ is satisfied when J_i becomes the head of KQ_x. A resource-holding job is ready.

K3 The head of KQ_x inherits the highest effective priority (which could be a donated priority) of any job in KQ_x.

K4 J_i is dequeued from KQ_x when $\mathcal{R}$ is completed. The new head of KQ_x, if any, acquires replica x.[5] J_i's priority donor (if any) may then issue a replica request subject to Rule D1.

Example 1. Consider a G-EDF-scheduled system on $m = 4$ processors with a single resource ℓ_a with $k_a = 2$ replicas as depicted in Fig 6. At time $t = 0$, jobs J_1 and J_2 are released with deadlines of $d_1 = 10$ and $d_2 = 13$. At time $t = 1$, J_3 and J_4 are released with deadlines $d_3 = 15$ and $d_4 = 14$. Also at $t = 1$, both J_1 and J_2 request and acquire a replica of ℓ_a. At time $t = 2$, J_3 requests a replica of ℓ_a, and is enqueued in KQ_2 and suspends by Rule K1. Then at time $t = 3$, J_5 and J_6 are released with deadlines of $d_5 = d_6 = 12$. At this time, J_1, J_4, J_5, and J_6 have the four highest effective priorities, and therefore are scheduled, even though J_2 is holding a replica of ℓ_a. At $t = 4$, J_4 requests a replica of ℓ_a, enqueues in KQ_1 and suspends by Rule K1. Also at $t = 4$, J_5 requests a replica of ℓ_a and because there are $m = 4$ jobs with incomplete replica requests, by Rule D2, J_5 must donate to J_3, the job with the

[5] As an implementation optimization, if KQ_x is left empty after J_i dequeues, a request from another KQ_y can migrate to KQ_x and acquire replica x to reduce average-case pi-blocking. For example, the highest effective-priority job could be chosen to migrate, or the job with the least remaining slack.

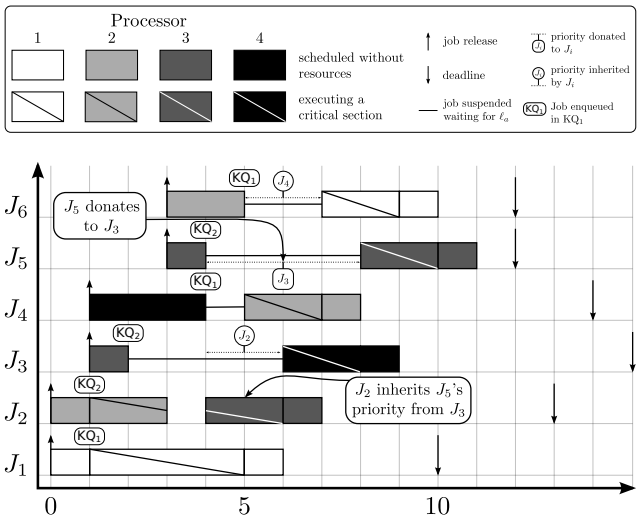

Figure 6: Figure depicting the task system in Example 1. In this example, $k = 2$ and $m = 4$.

lowest effective priority among the jobs with incomplete requests for a replica of ℓ_a. J_2 then inherits J_3's effective priority, or J_5's priority, by Rule K3. At time $t = 5$, J_1 releases its replica of ℓ_a, allowing J_4, the next job in KQ_1 to acquire a replica of ℓ_a. Also at time $t = 5$, J_6 requests a replica of ℓ_a, and is enqueued in KQ_1, J_4 also inherits J_6's priority by Rule K3. At time $t = 6$, J_2 releases its replica of ℓ_a, and J_3 acquires it. At time $t = 7$, J_4 releases its replica, which allows J_6 to begin its critical section. At $t = 8$, J_3 finishes its critical section, and J_5's donation obligation is finally completed by Rule D6, and it therefore is allowed to request a replica of ℓ_a. At this time, J_6 can immediately acquire its replica and begin its critical section. Finally, at times $t = 9$ and $t = 10$, J_6 and J_5 respectively complete their critical sections and the example returns to ordinary G-EDF scheduling.

Analysis. Next, we analyze the worst-case pi-blocking of the R^{2}DGLP. By Lemma 4, if a job is pi-blocked it either has an incomplete replica request or it is a priority donor. Thus, the total duration of pi-blocking is equal to the maximum duration of time a job can be pi-blocked while it is a priority donor as well as the maximum duration of time a job can be pi-blocked while it has an incomplete replica request. We analyze each of these times separately.

Lemma 6. *The maximum length of* KQ_x *is* $\lceil m/k \rceil$.

Proof. By Lemma 2, there are no more than m jobs with incomplete replica requests. By Rule K1, jobs are enqueued in the shortest queue upon request, and thus a job J_i will never be enqueued on a queue of length longer than $\lceil m/k \rceil$, otherwise there would have been a shorter queue on which J_i would have enqueued. □

Lemma 7. *Rule K3 ensures Property P1.*

Proof. If a job J_i with an incomplete request in KQ_x has sufficient effective priority to be scheduled, then by Rule K3, the job at the head of KQ_x inherits J_i's effective priority. Thus the replica holder is scheduled, and J_i makes progress. □

	Release-blocking	Request-blocking
R^{2}DGLP	0	$2(\lceil m/k \rceil - 1)L^{max}$
O-KGLP	0	$2(\lceil m/k \rceil + 2)L^{max}$
CK-OMLP	$\lceil m/k \rceil L^{max}$	$(\lceil m/k \rceil - 1)L^{max}$
k-FMLP	$\lceil n/k \rceil L^{max}$	$(\lceil n/k \rceil - 1)L^{max}$

Table 1: Blocking bounds of several k-exclusion suspension-based locking protocols.

Lemma 8. *A job* J_i *can be pi-blocked for* $(\lceil m/k \rceil - 1)L^{max}$ *in* KQ_x.

Proof. By Lemma 7, a job that is pi-blocked makes progress. By Lemma 6, there are at most $\lceil m/k \rceil - 1$ jobs that are enqueued ahead of J_i in KQ_x. Thus, the maximum duration of pi-blocking in KQ_x is $(\lceil m/k \rceil - 1)L^{max}$. □

Lemma 9. *A job* J_d *can be pi-blocked for a maximum duration of* $\lceil m/k \rceil L^{max}$ *while it is a priority donor.*

Proof. Follows from Lemmas 5 and 8. □

Theorem 1. *The maximum duration of pi-blocking a job* J_i *can experience waiting for a replica per request is* $(2\lceil m/k \rceil - 1)L^{max}$.

Proof. Follows from Lemma 4, 8, and 9. □

Note that the O-KGLP [9], the only other known asymptotically optimal k-exclusion locking protocol under s-oblivious analysis that does not cause release-blocking, has a worst-case pi-blocking bound of $(2\lceil m/k \rceil + 2)L^{max}$. Thus, the locking protocol we present has a worst-case blocking bound that improves upon the O-KGLP by $3L^{max}$. These blocking bounds can be seen in Table 1. As we show next in Sec. 5, this improvement can be quite significant when critical sections (i.e., L^{max}) are long.

Additionally, note that when $k = 1$, the blocking bound is $(2m - 1)L^{max}$, which is the same as that of the global OMLP [5] for serially reusable resources. The R^{2}DGLP is therefore a more flexible and applicable locking protocol in that it can be used either for a mutex lock, or a k-exclusion lock, both with good blocking bounds.

5 Experimental Results

To better understand the schedulability properties of the R^{2}DGLP, we randomly generated task sets with varying characteristics. Soft real-time schedulability under G-EDF scheduling was determined, as described in [12] for tasks with relative deadlines equal to periods ($d_i = p_i$). We focus our attention on soft real-time schedulability since global schedulers (the only type the R^{2}DGLP supports) are capable of ensuring bounded deadline tardiness in sporadic task systems with no utilization loss [16]. Schedulability was also tested under different locking protocols for compari-

son. These were the k-FMLP [11], the CK-OMLP [6], and the O-KGLP [9].

Experimental Setup. The task set characteristics varied by per-task utilization, number of replicas k, critical section length, and number of resource-using tasks in a task set. In all experiments, the system contained a single k-exclusion resource. *Utilization intervals* determine the range of utilization for individual tasks and were $[0.01, 0.1]$ (*light*), $[0.1, 0.4]$ (*medium*), and $[0.5, 0.9]$ (*heavy*). The number of replicas in each case varied among $k \in \{2, 4, 6, 8\}$. *Critical section intervals* determine the range of critical section lengths for resource-using tasks and were $(0\%, 2\%]$ (*very short*), $(0\%, 10\%]$ (*short*), $[10\%, 25\%]$ (*moderate*), and $[50\%, 75\%]$ (*long*), where critical section length is a percentage of e_i. The moderate and long intervals are inspired by GPU-usage patterns in which there are k GPUs [10] (a motivating example described in Sec. 1) while very short and short intervals may be common for other shared resources. *Resource usage percentage intervals* determine the number of tasks in a task set that use a resource protected by the k-exclusion lock and vary in increments of 10% from 0% to 100%. Each combination of these four parameters resulted in an experimental scenario. Each scenario was used to evaluate schedulability under each locking protocol on an eight CPU system. For example, one such scenario tested schedulability for task sets with *light* utilizations, $k = 4$ replicas, *short* critical section intervals, where 50% to 60% of tasks required the use of a replica of the shared resource. A total of 432 experimental scenarios were run.

We generated random task sets for each experimental scenario in the following manner. First, we selected a total system utilization cap uniformly in the interval $(0, 8]$ capturing the possible system utilizations on a platform with eight CPUs. We then generated tasks by making selections uniformly from the intervals in each scenario. Per-task utilization was selected from the scenario's utilization interval. Task periods were selected from the range $[3ms, 33ms]$, a common range for multimedia applications. Execution times were derived from the selected utilization and period. We added the generated tasks to a task set until the set's total utilization exceeded the utilization cap, at which point the last-generated task was discarded. Next, we designated some tasks to use the shared resource; we determined the number of resource-using tasks by selecting a percentage from the resource usage percentage interval of the scenario. A critical section length for each resource-using task was selected from the scenario's critical section interval. Bounds on pi-blocking were computed using detailed analysis such as that presented in [5, 4] for each tested locking protocol. As per s-oblivious analysis, task execution times were inflated by their respective pi-blocking bounds (i.e., $e_i^{inflated} = e_i + b_i$, where b_i is the pi-blocking bound of T_i), prior to performing the soft real-time schedulability test. Tardiness bounds were computed using the method developed by Erickson et al. since it offers the tightest know tardiness bounds for G-EDF schedulers [12]. These tardiness bounds were incorporated into fixed-point iterative schedulability tests. Fixed-point tests are necessary because tardiness can affect bounds on pi-blocking, which in turn can increase tardiness. Thus, tight tardiness bounds can improve soft real-time schedulability analysis.

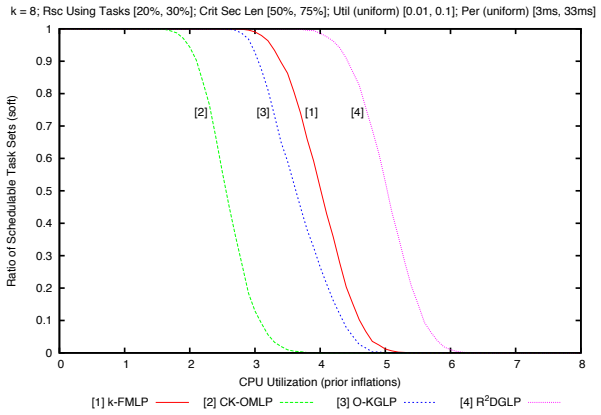

Figure 7: The R^{2}DGLP dominates both the CK-OMLP and O-KGLP. This scenario highlights that release-blocking affects negatively affect the CK-OMLP.

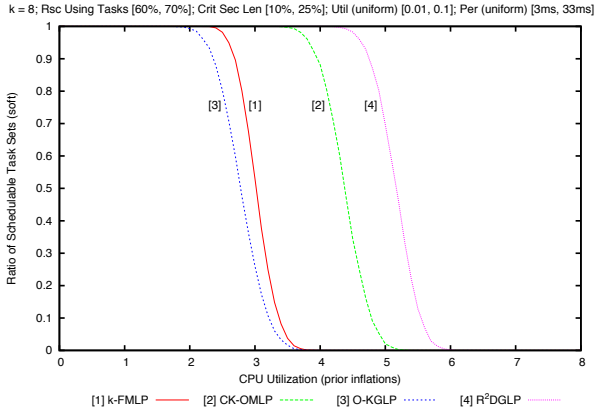

Figure 8: The R^{2}DGLP dominates both the CK-OMLP and O-KGLP. The O-KGLP performs poorly due to additional blocking for resource-using tasks

Results. A selection of results that demonstrate observable trends across all scenarios is presented here. We found that trends were most clearly expressed in scenarios using *light* utilizations since this resulted in task sets with more tasks.

***Observation* 1.** *Schedulability is poor under the CK-OMLP when critical section lengths are long and when there are relatively few resource-using tasks.*

Fig. 7 depicts schedulability under a scenario where a small percentage of tasks use a resource, yet each of these have long critical sections. Under the CK-OMLP, all non-resource-using tasks experience release-blocking from $\lceil m/k \rceil$ replica requests. This negatively affects schedulability in all cases, but is particularly harmful when critical

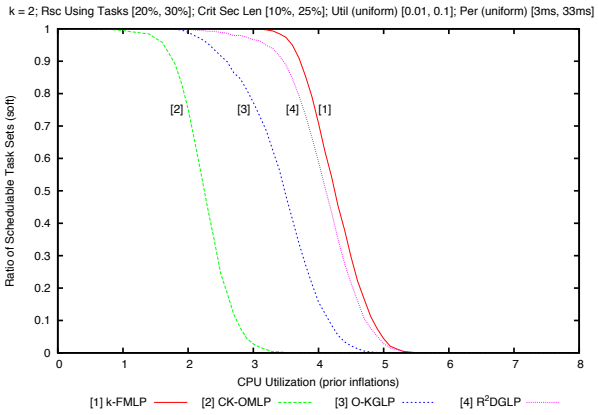

Figure 9: The R^{2}DGLP dominates both the CK-OMLP and O-KGLP, and the O-KGLP dominates the CK-OMLP. However, the k-FMLP outperforms the R^{2}DGLP (occurred in about 14% of tested scenarios).

section lengths are long, as is the case in Fig 7. Here, only 20% to 30% of tasks in every task set use a resource, but critical section lengths are long, ranging from 50% to 75% of execution time. Roughly 50% of task sets with utilizations of 5.0 are schedulable under the R^{2}DGLP. In contrast, approximately 50% of task sets with utilizations of 2.5 (half that of the R^{2}DGLP) are schedulable under the CK-OMLP. Further, no task sets with utilizations greater than 4.0 are schedulable under the CK-OMLP.

Observation 2. *The R^{2}DGLP improves upon the O-KGLP, especially when k is large.*

Fig. 7 also depicts schedulability under a scenario where k is large. Though both the R^{2}DGLP and O-KGLP are asymptotically optimal, the R^{2}DGLP significantly outperforms the O-KGLP. Under the R^{2}DGLP, resource-using tasks can experience request-blocking from $2\lceil m/k \rceil - 1$ other replica requests (Theorem 1). In comparison, these tasks experience request-blocking from $2\lceil m/k \rceil + 2$ other replica requests under the O-KGLP; three more requests than the R^{2}DGLP. When k is large, the addition of three requests to the blocking term has a stronger affect on schedulability. For example, in the scenario depicted in Fig. 7, $m = 8$ and $k = 8$, so under the R^{2}DGLP, requests are pi-blocked by only a single request versus four requests under the O-KGLP.

Observation 3. *Schedulability under the CK-OMLP can be better than the O-KGLP when there are many resource-using tasks.*

Fig. 8 illustrates a scenario where the number of resource-using tasks is relatively high and the CK-OMLP can outperform the O-KGLP. Here, we begin to see trade-offs between release-blocking and request-blocking between these two protocols. Like the R^{2}DGLP, resource-using tasks can experience pi-blocking from $2\lceil m/k \rceil - 1$ replica requests under the CK-OMLP, though non-resource-using tasks also experience pi-blocking (Obs. 1). Resource-using tasks experience pi-blocking from three additional requests under the O-KGLP, but non-resource-using tasks experience no pi-blocking. When the relative number of resource-using tasks is high, the blocking effects from non-resource-using tasks is decreased, while the blocking effects from resource-using tasks are magnified. Thus, neither the CK-OMLP or O-KGLP dominates the other in all scenarios. However, the R^{2}DGLP dominates both of these in all scenarios.

Observation 4. *The R^{2}DGLP strictly dominates the CK-OMLP and O-KGLP when jobs only make a single request.*

When jobs only make one replica request, the R^{2}DGLP offers the best schedulability of known optimal k-exclusion locking protocols for globally-scheduled JLFP systems. The dominance of the R^{2}DGLP over the CK-OMLP and O-KGLP in this case can be observed in Figs. 7, 8, and 9. The R^{2}DGLP exhibits the best aspects of both the CK-OMLP and O-KGLP: resource-using jobs can experience pi-blocking from $2\lceil m/k \rceil - 1$ replica requests, and non-resource-using jobs experience no pi-blocking.

Note that if individual jobs make many requests for shared resources than the CK-OMLP may have better schedulability. In the CK-OMLP, jobs can be blocked upon release, however, subsequent requests can only be blocked for $(\lceil m/k \rceil - 1)L^{max}$ instead of $(2\lceil m/k \rceil - 1)L^{max}$. Therefore, if jobs make many short requests, the CK-OMLP may be favorable to the R^{2}DGLP.

Observation 5. *The k-FMLP sometimes outperforms the R^{2}DGLP.*

Fig. 9 illustrates a scenario where the k-FMLP outperforms the R^{2}DGLP, despite not being asymptotically optimal. This occurred in about 14% of the tested scenarios. While the R^{2}DGLP is asymptotically optimal, there are cases where the k-FMLP can offer better schedulability. This is due to two aspects of the k-FMLP. First, resource-using tasks can experience pi-blocking from $\lceil n_a/k \rceil$ requests, where n_a is the number of tasks that may request ℓ_a. Thus, the k-FMLP can outperform the R^{2}DGLP when n_a is sufficiently small. Second, due to the FIFO ordering of all requests, a task can be pi-blocked by at most one request per task under the k-FMLP. Due to donation mechanisms, a task can be pi-blocked by at most two requests per task under the R^{2}DGLP, even though the total number of requests that may pi-block a task is $2\lceil m/k \rceil - 1$. A task may experience less pi-blocking under the k-FMLP if there is a high degree of variance in critical section lengths.

Minimum Response-Time Constraints. As discussed in Sec. 1, release-blocking imposes a minimum response time on all tasks, including those that do not use resources. This minimum response time is on the order of several of the longest critical sections. As a result, non-resource-using tasks must have similar response-time constraints to those of resource-using tasks. For implicit-deadline systems, this

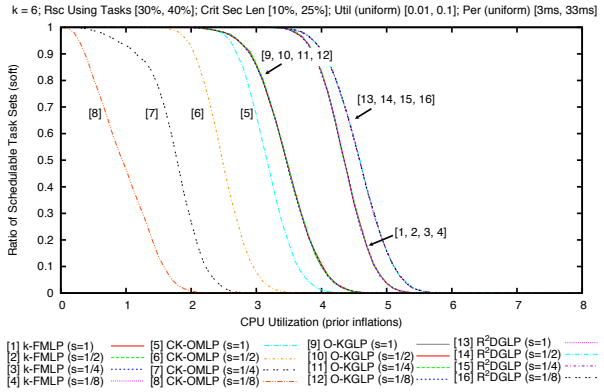

Figure 10: Release-blocking imposes a minimum response time on all tasks. This negatively affects schedulability of tasks with varying response-time constraints.

means that all tasks, resource-using and non-resource-using, must have similar periods. This limits system flexibility since not all potential applications have this characteristic. In order to highlight this limitation, we selected a scenario from our prior experiments and scaled the execution times and periods of *non*-resource-using tasks by a scaling factor, $s \in \{1/2, 1/4, 1/8\}$. Thus, task utilization remained constant, yet response-time constraints on non-resource-using tasks become more stringent with smaller scaling factors. The results of this are depicted in Fig. 10. It is easily observed that release-blocking negatively affects schedulability of the CK-OMLP. In contrast, the remaining protocols (which only incur request-blocking) are unaffected.

6 Conclusions

In this paper, we have presented RRPD, a progress mechanism for globally-scheduled multiprocessor real-time systems that does not cause release-blocking under s-oblivious analysis. Using a combination of RRPD and priority inheritance we have constructed the R^{2}DGLP, a k-exclusion locking protocol that is asymptotically optimal. The R^{2}DGLP improves upon existing k-exclusion locking protocols such as the O-KGLP and the clustered k-exclusion variant of the OMLP, as we have demonstrated. These improvements are particularly significant for applications in which critical sections are long, as is the case when using a locking protocol to arbitrate access to shared I/O devices such as GPUs.

The protocols that we have presented are applicable not only in globally scheduled systems, but also on a clustered systems for resources that are shared locally within a single cluster. In the future, we hope to investigate clustering algorithms, in which the cluster sizes are determined, and tasks assigned to clusters in such a way as to minimize sharing across clusters. This allows for more lightweight progress mechanisms such as RRPD to be employed instead of more heavyweight progress mechanisms such as JRPD, which cause more release-blocking.

References

[1] T.P. Baker. A stack-based resource allocation policy for real-time processes. In *RTSS '90*, pages 191–200, 1990.

[2] A. Bastoni, B.B. Brandenburg, and J.H. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor real-time schedulers. In *RTSS '10*, 2010.

[3] 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.

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

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

[6] 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.

[7] U.M.C. Devi and J.H. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. In *RTSS '05*, pages 133–189, Dec. 2005.

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

[9] 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.

[10] G.A. Elliott and J.H. Anderson. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems*, 48:34–74, 2012.

[11] G.A. Elliott and J.H. Anderson. Robust real-time multiprocessor interrupt handling motivated by GPUs. In *ECRTS '12*, 2012 (to appear).

[12] J. Erickson, U. Devi, and J. Anderson. Improved tardiness bounds for Global EDF. In *ECRTS '10*, pages 14–23, 2010.

[13] P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *RTSS '01*, 2001.

[14] P.-C Hsiu, D.-N Lee, and T.-W Kuo. Task synchronization and allocation for many-core real-time systems. In *EMSOFT '11*, 2011.

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

[16] H. Leontyev and J. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. *Real-Time Systems*, 44(1):26–71, 2010.

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

[18] B.C. Ward and J.H. Anderson. Supporting nested locking in multiprocessor real-time systems. In *ECRTS '12*, 2012 (to appear).