An Optimal k-Exclusion Real-Time Locking Protocol Motivated by Multi-GPU Systems[*]

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

Abstract

Graphics processing units (GPUs) are becoming increasingly important in today's platforms as their increased generality allows for them to be used as powerful co-processors. In previous work, we have found that GPUs may be integrated into real-time systems through the treatment of GPUs as shared resources, allocated to real-time tasks through mutual exclusion locking protocols. In this paper, we present an optimal k-exclusion locking protocol for globally-scheduled job-level static-priority (JLSP) systems. This protocol may be used to manage a pool of GPU resources in such systems.

1 Introduction

The widespread adoption of multicore technologies in the computing industry has prompted research in a wide variety of computing fields with the goal of better understanding how to exploit multicore parallelism for greater levels of performance. In the field of real-time systems, multicore technologies have led to the revisiting of problems that have had well understood uniprocessor solutions. This research has found that uniprocessor techniques are often no longer valid or suffer from significant inefficiencies when applied directly to multiprocessor platforms. As a result, new algorithms for scheduling and synchronization and new methods of analysis have been developed. However, the topic of k-exclusion synchronization has only recently been considered for real-time multiprocessor applications [4]. k-exclusion locking protocols can be used to arbitrate access to pools of similar or identical resources, such as communication channels or I/O buffers. k-exclusion extends ordinary mutual exclusion (mutex) by allowing up to k tasks to simultaneously hold locks (thus, mutual exclusion is equivalent to 1-exclusion). In this paper, we present a new protocol for implementing k-exclusion locks in multiprocessor real-time systems.

The commonality of resource pools is enough to motivate our investigation of k-exclusion protocols for such systems. However, we are specifically driven to study such protocols due to their application to another new technology: general-purpose computation on graphics processing units (GPGPU). In prior work, we showed that mutual exclusion locks may be used to integrate individual GPUs into real-time multiprocessor systems.

The use of mutual exclusion locks naturally complements the state of current GPU technology and resolves many technical challenges arising from both hardware and software constraints [6], thus allowing guarantees on predictable execution as required by real-time systems. For example, current technology does not allow the preemption of program code executing on a GPU. Thus, once a task has commenced using a GPU, it must finish using the GPU before another task may make use of it. This can easily result in priority inversions with respect to CPU scheduling. From this perspective, there is little to distinguish the GPU, which is an I/O device, from a traditional resource. The use of a real-time locking protocol may be used to bound the durations of priority inversions.

In addition to the fundamental limitations imposed by non-preemption, GPUs (because they are I/O devices) are managed by closed-source device drivers that are currently not designed for real-time applications. This introduces a number of challenges [6]. However, the use of mutual exclusion locks to arbitrate access to a GPU prevents the closed-source driver from invoking its own resource arbitration mechanisms (such as unbounded spinning on the CPU). Thus, the use of real-time mutual exclusion locks with GPUs simultaneously addresses issues caused by non-preemption and largely prevents the GPU driver from introducing non-real-time behaviors.

This resource-based methodology can be extended to systems with multiple GPUs through the use of k-exclusion locks to protect pools of GPU resources. Such an approach may maximize GPU utilization as it avoids the need to statically assign real-time tasks to use individual GPUs. In this

[*] Work supported by NSF grants CNS 0834270, CNS 0834132, and CNS 1016954; ARO grant W911NF-09-1-0535; AFOSR grant FA9550-09-1-0549; and AFRL grant FA8750-11-1-0033.

paper, we present an optimal k-exclusion protocol that can be used to realize this approach in globally-scheduled real-time systems. At this time, we see GPU computation to be more relevant to soft real-time computing than hard real-time computing, due to difficult unresolved timing analysis issues affecting the latter on multicore platforms. Thus, our focus on global scheduling is motivated by the fact that a variety of global schedulers are capable of ensuring bounded deadline tardiness with no utilization loss [9]. Such schedulers are particularly well-suited for supporting soft real-time workloads. However, this attention to soft real-time applications does not preclude the use of the locking protocol presented in this paper in systems with hard real-time constraints.

Prior Work. k-exclusion locking protocols for real-time systems has been investigated before. Chen [5] presented techniques to adapt several common uniprocessor mutex protocols to derive uniprocessor k-exclusion locks. However, the use of such techniques in a multiprocessor environment requires that tasks and resources be statically bound to individual processors. This static partitioning may place undesirable limits on maximum system utilization. Further, optimal partitioning is NP-hard in the strong sense [10], even without consideration of resource locality.

Much more recently, Brandenburg et al. presented an extension to the *O(m) Locking Protocol* (OMLP) to support k-exclusion locks on cluster-scheduled multiprocessors [4].[1] While the k-OMLP may be applied to globally-scheduled systems (since a globally-scheduled system is a degenerate case of a cluster-scheduled system where there is only one cluster), real-time tasks not requiring one of the k resources may still experience delays in execution. These delays are artifact behaviors that are required in clustered scheduling, but not necessarily in global scheduling. Such delays may not be particularly harmful, in terms of schedulability, when the k-OMLP is used to protect resources with short protection durations (as may be the case with internal data structures), but may be extremely detrimental in systems using GPUs. This is due to the fact that protection durations (i.e., *critical section lengths*) for GPU resources may be very long, on the order of tens of milliseconds to even several seconds [6]. Thus, it is desirable to develop a k-exclusion protocol for globally-scheduled systems that does not affect the execution of non-GPU-using tasks. Note that such a protocol can be applied in a clustered setting if GPUs are statically allocated to clusters (in which case, clusters can be scheduled independently).

To find inspiration for an efficient k-exclusion locking protocol for real-time systems, one may also look at k-exclusion protocols from the distributed algorithms literature, where research has been quite thorough (see, e.g., [1, 13]). However, such protocols were designed for the use in throughput-oriented systems for which predictability is not a major concern.

The use of resource locks is not the only approach that may be taken to integrate GPUs into a real-time system. A heterogeneous processor-scheduling algorithm may schedule dissimilar processors. However, existing approaches, such as [2, 8, 12], are problematic in multi-GPU systems due to one or more of the following constraints: (i) they cannot account for non-preemptive GPU execution; (ii) they require that tasks be partitioned among types of processors yet our GPU-using tasks must make use of both CPU and GPU processors; (iii) they statically assign GPU-using tasks to GPUs; (iv) they place restrictions on how GPUs may be shared among tasks; and (v) they place limits on the number of CPUs and GPUs. Heterogeneous processor-scheduling approaches may better express the parallel execution of CPU and GPU code (which is not captured when GPUs are treated as shared resources instead of processors), though they may place too many constraints on some systems. Of course, further research in this direction is clearly merited, though we leave this to future work.

Contributions. In this paper, we present a new real-time k-exclusion locking protocol for globally-scheduled real-time multiprocessor systems. This protocol is asymptotically optimal under suspension-oblivious schedulability analysis [3]. Our protocol is designed with real-time multiprocessor systems with multiple GPUs in mind. This leads us to use techniques that (i) minimize the worst-case wait time a task experiences to receive a resource, as this helps meet timing constraints; (ii) do not cause non-resource-using tasks to block; (iii) yield beneficial scaling characteristics of worst-case wait time with respect to resource pool size, since pool size directly affects system processing capacity in the GPU case; and (iv) increase CPU availability through the use of suspension-based methods, which aids in meeting timing constraints in practice. While our focus is on GPUs as resources, our protocol may still be used to efficiently manage pools of generic resources, offering improvements over the k-OMLP on globally-scheduled systems.

Organization. The rest of this paper is organized as follows. In Sec. 2, we describe the task model upon which our locking protocol is built. In Sec. 3, we discuss what it means for a locking protocol to be "optimal" in a globally scheduled system and how it might be achieved. In Sec. 4, we discuss how even an informed approach can lead to sub-optimal characteristics in a k-exclusion locking protocol. We also

[1] To the best of our knowledge, this is the first work investigating the k-exclusion problem in globally-scheduled real-time multiprocessor systems.

present our k-exclusion locking protocol and prove its optimal characteristics in this same section. We end in Sec. 5 with concluding remarks and avenues for future work.

2 Task Model

We consider the problem of scheduling a mixed task set of n sporadic tasks, $T = \{T_1, \ldots, T_n\}$, on m CPUs with one pool of k resources. A subset $T^R \subset T$ of the tasks require use of one of the system's k resources. We assume $k \leq m$. A *job* is a recurrent invocation of work by a task, T_i, and is denoted by $J_{i,j}$ where j indicates the j^{th} job of T_i (we may omit the subscript j if the particular job invocation is inconsequential). Each *task* T_i is described by the tuple $T_i(e_i, l_i, d_i, p_i)$. The *worst-case CPU execution time* of T_i, e_i, bounds the amount of CPU processing time a job of T_i must receive before completing. The *critical section length* of T_i, l_i, denotes the length of time task T_i holds one of the k resources. For tasks $T_i \notin T^R$, $l_i = 0$. The *deadline*, d_i, is the time after which a job is released by when that job must complete. Arbitrary deadlines are supported in this work. The *period* of T_i, p_i, measures the minimum separation time between job invocations for task T_i.

We say that a job J_i is *pending* from the time of its release to the time it completes. A pending job J_i is *ready* if it may be scheduled for execution. Conversely, if J_i is not ready, then it is *suspended*. Throughout this paper, we assume that the tasks in T are scheduled using a job-level static-priority (JLSP) global scheduler.

A job $J_{i,j}$ (of a task $T_i \in T^R$) may issue a resource request $R_{i,j}$ for one of the k resources. Requests that have been allocated a resource (*resource holders*) are denoted by H_x, where x is the index of the particular resource (of the k) that has been allocated. Requests that have not yet been allocated a resource are *pending* requests. Motivated by common GPU usage patterns, we assume that a job requests a resource at most once, though the analysis presented in this paper can be generalized to support multiple, non-nested, requests. We let b_i denote an upper bound on the duration a job may be blocked.

In this paper, we consider locking protocols where a job J_i suspends if it issues a request R_i that cannot be immediately satisfied. In such protocols, priority-sharing mechanisms are commonly used to ensure bounded blocking durations. *Priority inheritance* is a mechanism where a resource holder may temporarily assume the higher priority of a blocked job that is waiting for the held resource. Another common technique is *priority boosting*, where a resource holder temporarily assumes a maximum system scheduling priority. The priority of a job J_i in the absence of priority-sharing is the *base priority* of J_i. We call the priority with which J_i is scheduled the *effective priority* of J_i.

3 Definition of Optimality

Generally speaking, a job of a real-time task is *blocked* from execution when it attempts to acquire a resource from some set of resources of which there are none currently available; the job must wait until said resource becomes available. Schedulability analysis requires that these blocking durations be of bounded length to ensure that timing constraints, such as completing by a given deadline, are met. In [3], this definition of blocking was refined for JLSP globally-scheduled multiprocessor systems, allowing for a definition of optimality in blocking duration to be made.

It was observed that a real-time job is "blocked" only if it waits for a resource when it would otherwise be scheduled. When a job lacks sufficient priority to be scheduled, it makes no difference in terms of analysis if it is suspended implicitly by the scheduler or if it is suspended while waiting for a resource. The effect is the same: the job is not scheduled. It is only the duration of time that a job would be scheduled, but otherwise cannot due to waiting, that must be considered by analysis. In such cases there is a *priority inversion* since a lower-priority job may be scheduled in the blocked job's place. Thus, this refined definition of blocking is termed *priority inversion blocking*, or *pi-blocking*. The method to bound the time a job may experience pi-blocking depends upon the scheduling algorithm used and its analysis.

Assuming jobs suspend from execution (instead of busy-waiting) while waiting for a resource, the analytical method used to determine the effect of pi-blocking may be *suspension-oblivious* or *suspension-aware*. Suspension-oblivious analysis treats delays caused by pi-blocking as additional execution time, factoring into task utilization and thus into task set utilization as well. This treatment converts a set of dependent tasks into a task set of independent tasks with greater execution requirements. This is a safe conversion, but may be pessimistic if pi-blocking delays are long. In contrast, suspension-aware analysis does not treat pi-blocking delays as processor demand. Unfortunately, most known multiprocessor schedulability analysis techniques for JLSP global schedulers, such as the *global earliest-deadline-first* (G-EDF) algorithm, that account for blocking delays are suspension-oblivious.

It was shown in [3] that under suspension-oblivious analysis, a job J_i is not pi-blocked if there exist at least m pending higher-priority jobs, where m is the number of system CPUs. Because suspensions are analytically treated as execution time under suspension-oblivious analysis, even suspended jobs of higher-priority can eliminate priority inversions with respect to lower-priority jobs. If it can be shown in the analysis of a locking protocol that there exist at least m higher-priority suspended jobs that are waiting for a resource, then lower-priority jobs also waiting for a resource *do not experience any pi-blocking*. Such an example is illus-

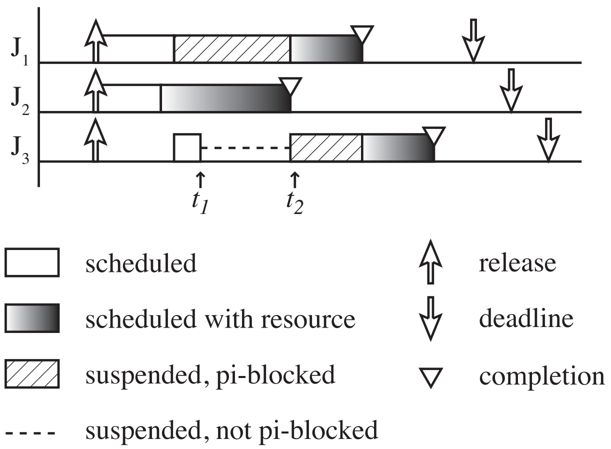

Figure 1. Job J_3 does not experience pi-blocking on the interval $[t_1, t_2]$ under suspension-oblivious analysis for this two-processor system scheduled by G-EDF. Job J_2 is scheduled on this interval while job J_1 is *analytically considered* to be scheduled. Job J_3 is not pi-blocked because it does not have sufficient priority to be scheduled (analytically), whether it waits for a resource or not.

trated in Fig. 1 for a two-processor system scheduled under G-EDF with a single shared resource. As depicted, the presence of pending jobs J_1 and J_2 on the interval $[t_1, t_2]$ prevents J_3 from incurring any pi-blocking under suspension-oblivious analysis.

The OMLP, as well as the locking protocol presented in this paper, are specifically designed to exploit this characteristic of suspension-oblivious analysis. Through this analysis, it was further shown in [3] that a mutex locking protocol may be considered optimal under suspension-oblivious analysis if the maximum duration of pi-blocking per resource request is O(m)—a function of fixed system resource parameters and not the number of resource-using tasks. In a k-exclusion locking protocol, we may hope to do better. Intuitively, we would like to obtain a bound of O(m/k), so pi-blocking durations scale with the inverse of k (another fixed system parameter). Indeed, the k-OMLP achieves this bound when there exists only one pool of k resources, as is the case with our GPU system. However, as stated earlier, the k-OMLP is not suitable for our use on a JLSP globally-scheduled system with GPUs due to the excessive blocking costs charged to non-GPU-using tasks. Still, any efficient k-exclusion locking protocol we develop for a JLSP globally-scheduled system should be O(m/k).

4 Locking Protocols

Developing an efficient k-exclusion protocol for JLSP globally-scheduled systems is a non-trivial process. Through the development of an O(m/k) k-exclusion locking protocol, we found that some initial assumptions did not hold. We will now explain the development process we went through to arrive at an optimal k-exclusion locking protocol.

4.1 A Single Queue

A classic result from Operations Research states that a single wait-queue is the most efficient method for ordering resource requests for a pool of resources [11]. Without presenting the details of this result, we may come to understand this to be true intuitively. Consider the case where a separate queue is used for each resource. There may exist an "unlucky" request, R_i, that is enqueued behind a job that uses a resource for a very long duration. In the meantime, other requests, including those made after R_i, are quickly processed on the other queues, yet the unlucky request continues to wait. To make a colloquial analogy, this is much like the frustration one may feel at the checkout line in a grocery store. You may find yourself stuck behind someone who needs a dozen price checks on their items, while you watch others quickly pass through the remaining lines. It is impossible for a request to be forced to wait on a long-running job when a single queue is used. Hence, the single queue reduces overall wait time for all participants.

The Bank Algorithm [7] (not to be confused with Dijkstra's Banker's Algorithm) is a non-real-time k-exclusion locking protocol built upon the single-queue principle. It is so named due to its likeness to the single queue commonly used at a bank. Suppose we built a real-time locking protocol based upon the Bank Algorithm. There would be one FIFO queue for k resources. We can ensure no pending request is blocked unboundedly through priority inheritance. For our real-time Bank Algorithm, let each of the k resource holders (if that many exist) inherit a unique priority, if that priority is greater than its own, from the set of the k highest-priority pending requests (if that many exist). Thus, at least one resource holder is scheduled with an effective priority no less than that of any pending request. In the worst-case scenario for the highest-priority pending request, R_i, all pending resource requests ahead of R_i are serialized through a single resource, while the remaining $k-1$ resources remain held. This may occur since these $k-1$ resource holders do not inherit a priority from R_i and may not be scheduled. This case is depicted in Fig. 2. With a little work, it is possible to combine methods from the Bank Algorithm and the OMLP to arrive at an $\Omega(m-k)$ locking protocol. However, this still falls short of our desired O(m/k).

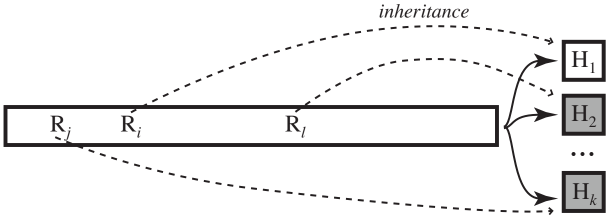

Figure 2. Pending requests ahead of R_i may be serialized through a single resource when a single wait-queue is used. Depicted above, R_i is the highest-priority request and resource holder H_1 inherits R_i's priority. The remaining $k-1$ resource holders inherit priority from pending requests with priorities less than R_i. The resource holders that do not inherit from R_i are not guaranteed to release their resource before R_i acquires one, thus all pending requests ahead of R_i may be forced to serialize on the resource held by H_1. In the worst-case, R_i may have to wait for $n-k$ requests to complete before obtaining a resource.

In a non-real-time context, it is implicitly assumed that all resource holders execute simultaneously. However, this guarantee cannot be maintained in our real-time system since the priority of the highest-priority job can only be inherited by a single resource holder.[2] It appears that for traditional sporadic real-time systems, a single queue approach will not yield an optimal bound for worst-case pi-blocking time because it is possible for resource requests to become serialized on a single resource. We must develop a k-exclusion locking protocol where execution progress can be guaranteed for all resource holders.

4.2 An Optimal k-Exclusion Global Locking Protocol

The *Optimal k-Exclusion Global Locking Protocol* (O-KGLP) is a k-exclusion locking protocol that achieves the desired O(m/k) bound. In the previous section, we noted that a straightforward application of OMLP techniques to the k-exclusion problem results in $\Omega(m-k)$ pi-blocking time. While this is optimal with respect to the number of CPUs, it does not fully exploit the greater parallelism offered by the existence of k resources. The O-KGLP offers better scaling behavior with respect to both the number of processors and resources.

Structure. The O-KGLP uses $k+1$ job queues to organize resource requests. k FIFO queues, of length m/k, are assigned to each of the k resources. One priority queue (ordered by job priority) is used if there are more than m jobs contending for the use of a protected resource. The priority queue holds the "overflow" from the fixed-capacity FIFO queues. We denote the FIFO queues as FQ$_x$ and the priority queue as PQ.

Rules. Let $queued(t)$ denote the total number of queued jobs in the PQ and FQs at time t. The rules governing queuing behavior and priority inheritance are as follows:

O1 When job J_i requests a resource at time t_0,

- **O1.1** R_i enqueues on the shortest FQ$_x$ if $queued(t_0) < m$, else
- **O1.2** R_i is added to PQ.

O2 All queued jobs are suspended except the jobs at the heads of the FQs, which are resource holders. All resource holders are ready to execute.

O3 The effective priority of a resource holder, H_x, at time t is inherited from either the highest-priority request in FQ$_x$, or from a unique request among the k highest-priority pending requests in the PQ, whichever has greater priority. Each FQ claims one unique request (if available) from the k highest-priority pending request in PQ, whether or not H_x inherits priority from it.

O4 When H_x frees a resource, its request is dequeued from FQ$_x$ and the next request in FQ$_x$, if one exists, is granted the newly available resource.[3] Further, the claimed unique request (if it exists) from amongst the k highest-priority requests in the PQ is moved to FQ$_x$.

Let us define several simplifying identifiers. Let PQ^{HP} (for "high priority") denote the set of $\min(k, |PQ|)$ highest-priority pending requests in the PQ. Let U_x denote the unique request in PQ^{HP} associated with H_x by Rule O3. Finally, let PQ^{LP} (for "low priority") denote the set of requests in PQ that are not in PQ^{HP}. Fig. 3 depicts the queue structure of the O-KGLP and inheritance relations.

In our initial analysis of the O-KGLP, we make the following assumption:

A1 U_x is never evicted from PQ^{HP} by the arrival of new, higher-priority, requests.

[2] We have considered algorithms where a single priority is inherited by multiple resource holders. However, we found that this breaks the sporadic task model since multiple jobs may execute concurrently with the same inherited priority. Different schedulability tests are required to analyze such a method.

[3] As an implementation optimization, if FQ$_x$ is left empty by the dequeue of H_x, then the highest-priority pending request in the remaining FQs may be "stolen" (removed from its queue and enqueued onto FQ$_x$) and granted the free resource, if such a request exists. This technique may reduce the observed average time jobs are blocked in a real system, but does not improve upon the worst case.

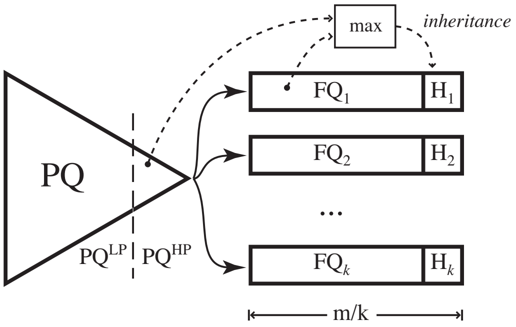

Figure 3. Queue structure and priority inheritance relations used by the O-KGLP.

This important property is guaranteed by ensuring that U_x always has a sufficient *effective priority* to remain in PQ^{HP}. The mechanisms used to realize Assumption A1 are explained in detail later in this paper.

Before bounding the worst-case pi-blocking time a job using the O-KGLP may experience, let us define the term *progress*. We say a pending request R_i makes *progress* at time instant t if every H_x, ahead of R_i on any path through the queues that R_i may take before obtaining a resource, is scheduled with an effective priority no less than that of R_i. If R_i is pi-blocked for a bounded time b_i, then R_i is no longer pi-blocked after b_i time of progress.

Progress is ensured with relative ease through priority sharing mechanisms (inheritance, boosting, etc.) in common locking protocols where a request can only follow a single path. However, progress is more difficult to ensure when more than one path may be taken, as is the case in the O-KGLP due to its use of k FQs. We now explain how this is done in the O-KGLP.

Blocking Analysis. J_i may be pi-blocked during three different phases as its request traverses the queues in the O-KGLP. The first phase is the duration from when R_i enters the PQ until it joins the set PQ^{HP}. The second phase takes place from the time R_i joins PQ^{HP} to the time it is moved to an FQ. Finally, the last phase is measured from the time R_i enqueues on an FQ to the point R_i reaches the head of this FQ. We denote pi-blocking in each phase as b^{LQ}, b^{HQ}, and b^{FQ}, respectively. The worst-case time J_i may be pi-blocked using the O-KGLP is equal to the sum of the maximum pi-blocking durations in each phase: $b_i = b_i^{LQ} + b_i^{HQ} + b_i^{FQ}$. These phases are depicted in Fig. 4.

The number of tasks, $\left|T^R\right|$, using the same O-KGLP lock determines whether a job may experience blocking in each of these three phases. For example, if $\left|T^R\right| \le k$, then no job is ever pi-blocked ($b_i = 0$) since every request can be trivially satisfied simultaneously. If $k < \left|T^R\right| \le m$, then a job only experiences b^{FQ} pi-blocking since all possible simultaneous requests can be held in the FQs. Similarly, b^{FQ} and b^{HQ} contribute to total pi-blocking when $m < \left|T^R\right| \le m + k$. A job can only experiences pi-blocking in every phase when $\left|T^R\right| > m + k$. Let us compute the worst-case pi-blocking a job J_i may experience starting with b^{FQ} and working our way backwards through the queue structures.

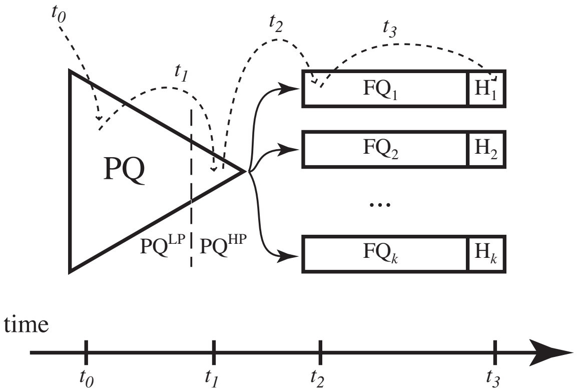

Figure 4. Job J_i may experience pi-blocking in three time intervals: first in the interval $[t_0, t_1)$, from when J_i's request, R_i, enters the PQ to when the request joins the set PQ^{HP}; next, in the interval $[t_1, t_2)$, which is the duration R_i is in PQ^{HP}; and finally, in the interval $[t_2, t_3)$, which is the time R_i must wait in an FQ until it receives a resource.

Lemma 1. *A job J_i may be pi-blocked by at most $\min\left(\frac{m}{k} - 1, \left\lfloor \frac{\left|T^R\right|-1}{k} \right\rfloor\right)$ lower-priority jobs while enqueued on* FQ$_x$.

Proof. Progress is ensured for any R_i in FQ$_x$ since H_x is always scheduled with an effective priority no less than J_i by Rule O3. Thus b_i^{FQ} can be bounded by the total time required to complete every request ahead of R_i in FQ$_x$.

In the worst case, while R_i is on FQ$_x$, it may be preceded by $\frac{m}{k} - 1$ requests before it reaches the head of FQ$_x$ and J_i receives a resource. However, if $k < \left|T^R\right| \le m$, then J_i may be pi-blocked by fewer requests. In this case there may be as many as $\left|T^R \backslash \{T_i\}\right|$ requests already in the FQs when J_i issues R_i at time t_0. Load-balancing these preceding requests evenly across the k FQs (by Rule O1.1), the shortest FQ at time t is at most $\left\lfloor \frac{\left|T^R \backslash \{T_i\}\right|}{k} \right\rfloor$ in length since the length of any FQ may only deviate from the average FQ length by more than one. Thus, $\left\lfloor \frac{\left|T^R\right|-1}{k} \right\rfloor$ upper-bounds the number of lower-priority jobs that may pi-block J_i when $k < \left|T^R\right| | \le m$. □

Lemma 2. *J_i experiences pi-blocking while R_i is queued on* FQ_x *of at most*

$$b_i^{FQ} = \min\left(\frac{m}{k} - 1, \left\lfloor \frac{|T^R| - 1}{k} \right\rfloor\right) \cdot l^{max} \tag{1}$$

where l^{max} denotes the longest critical section of any task.

Proof. J_i experiences worst-case pi-blocking when the jobs that pi-block it have the longest possible critical sections. By Lemma 1 and by upper-bounding critical section lengths with l^{max}, the proof follows. □

Lemma 3. *J_i may only be pi-blocked for the duration of one critical section while its request is in the set PQ^{HP}. Thus,*

$$b_i^{HQ} = l^{max} \tag{2}$$

in the worst case.

Proof. By Assumption A1, a request R_i cannot be evicted from PQ^{HP}. By Rule O3, there exists some FQ_x such that H_x is scheduled with an effective priority no less than that of J_i while R_i is in PQ^{HP}, thus progress is guaranteed. Further, R_i will be removed from the PQ and placed onto FQ_x immediately after H_x releases its resource. It may take up to l^{max} time until H_x complete its critical section, thus $b_i^{HQ} = l^{max}$. □

We now present a derivation of b_i^{LQ} by placing an upper bound on the number of lower-priority jobs that may pi-block J_i while R_i is in the PQ and not in PQ^{HP}. Recall from Sec. 3 that a job is not pi-blocked at any time instant under (suspension-oblivious analysis) if there exist at least m pending higher-priority jobs.

Lemma 4. *Progress is guaranteed for any request, R_i, pending in PQ^{LP}.*

Proof. Assumption A1 ensures that each $U_x \in \text{PQ}^{HP}$ has a priority no less than that of $R_i \in \text{PQ}^{LP}$. Thus, by Rule O3, each H_x is scheduled with an effective priority greater than R_i while $R_i \in \text{PQ}^{LP}$. Hence, progress for R_i is guaranteed for any path that R_i may take, even though the particular FQ R_i will traverse has yet to be determined. □

Lemma 5. *Job J_i, with request $R_i \in PQ^{LP}$, is pi-blocked for at most $\frac{m}{k} \cdot l^{max}$ time. Thus,*

$$b_i^{LQ} = \frac{m}{k} \cdot l^{max}. \tag{3}$$

Proof. A job is not pi-blocked under suspension-oblivious analysis when there exist at least m other pending higher-priority jobs. By Lemma 4, all the resource holders in the FQs are scheduled with an effective priority at least that of R_i while $R_i \in \text{PQ}^{LP}$. Consequently, all potential lower-priority requests in the FQs when J_i issued R_i at time t_0 (see Fig. 4) will be satisfied in at most $\frac{m}{k} \cdot l^{max}$ time if R_i continues to remain in PQ^{LP}. If R_i is in PQ^{LP} after $t_0 + \frac{m}{k} \cdot l^{max}$ time (which is possible since new requests with a higher priority than R_i may be issued before R_i is moved to PQ^{HP}), then, by Rule O4 (and Assumption A1), all m requests in the FQs must have a higher priority than R_i, and J_i is no longer pi-blocked. □

It may appear that we have arrived at an O(m/k) k-exclusion locking protocol since each component of b_i is either O(m/k) (b_i^{FQ} and b_i^{LQ}) or O(1) (b_i^{HQ}). However, our proofs for these bounds are founded upon the assumption that each request, once in PQ^{HP}, remains so until it is moved to an FQ. Our bound for b_i^{HQ} breaks if we allow evictions. Consider the following scenario, which is depicted in Fig. 5.

Suppose at time t_0, H_x has just received a resource and H_x inherits the priority of R_i, the highest priority request in the PQ. At time $t_1 = t_0 + (l^{max} - \varepsilon_0)$, k new requests with priorities greater than R_i are issued, and R_i is evicted from PQ^{HP}. At time $t_2 = t_1 + \varepsilon_1$, H_y completes and releases its resource. Consequently, one of the new requests is moved to FQ_y and R_i rejoins the set PQ^{HP}. By Rule O3, R_i is claimed by H_y, though H_y does not inherit the priority of R_i since the newer request that just entered FQ_y has greater priority. At this point, the $l^{max} - \varepsilon$ progress R_i had accrued before eviction has been lost.

Still, perhaps the *number* of times R_i can be evicted, *while* R_i remains pi-blocked, can bounded by an O(m/k) term in a similar fashion to b_i^{LQ}. After all, it seems reasonable that higher-priority requests should be able to enter the FQs ahead of R_i. Unfortunately, so may lower-priority request. Continuing the scenario above (soon after R_i has rejoined PQ^{HP}), at time $t_3 = t_2 + \varepsilon_2$ all resource holders except H_y complete and the requests of $\text{PQ}^{HP} \backslash R_i$ (which have higher priority than R_i) are moved to the FQs, and requests with priorities less than R_i join PQ^{HP}. At time $t_4 = t_3 + \varepsilon_3$, once again all resource holders except H_y complete, only now lower-priority requests are moved onto the FQs and R_i remains in PQ^{HP}. Finally, at time $t_5 = t_4 + \varepsilon_4$, another batch of new k higher-priority requests is issued, evicting R_i from PQ^{HP} once again. This cycle may repeat with requests of lower priority than R_i entering any FQ, so we cannot prove the presence of m pending higher-priority jobs as is required

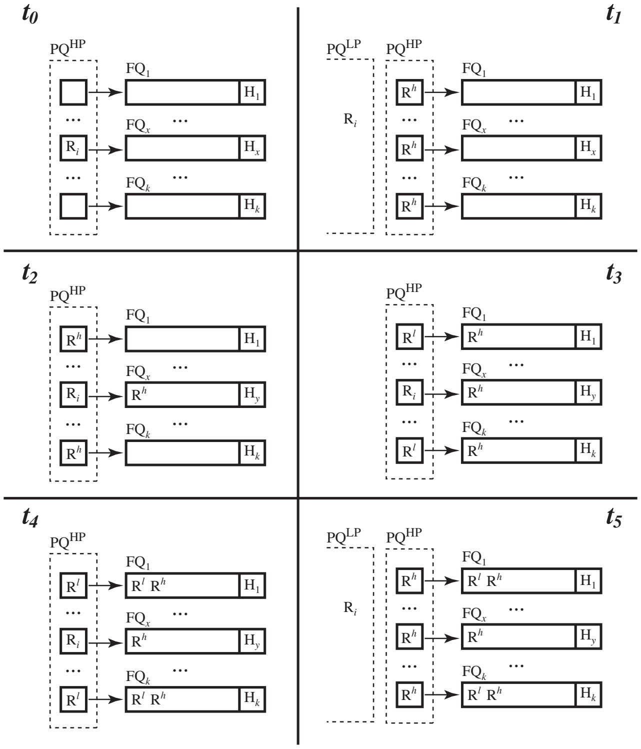

Figure 5. Unbounded pi-blocking for b_i^{HQ} if evictions from PQ^{HP} are allowed. t_0: R_i is in PQ^{HP} and H_x inherits the priority of R_i. t_1: R_i is evicted from PQ^{HP} by the arrival of k higher-priority requests. t_2: Resource holder H_y completes; R_i rejoins PQ^{HP}. t_3: All resource holders other than H_y complete. t_4: Once again, all resource holders other than H_y complete. t_5: R_i is evicted once again from PQ^{HP} by the arrival of k higher-priority requests. There are lower-priority requests in FQs (except for FQ$_y$, which may through repetitions of this scenario), so we cannot bound the time R_i is pi-blocked while in PQ^{HP}.

to end pi-blocking under suspension-oblivous analysis.[4]

Merely disallowing PQ^{HP} evictions will not resolve these issues since doing so trades one k-exclusion problem (resources) for another (the FQs). Since we cannot disallow the arrival of new higher-priority requests, another mechanism is required to maintain our O(1) bound for b_i^{HQ}. We will introduce three additional rules inspired by *priority donation* to maintain A1.

Developed by Brandenburg et al. [4], priority donation is priority inheritance technique that allows for the bounding of pi-blocking on cluster-scheduled systems. At a high level, jobs with higher priorities may temporarily suspend and donate their priority to resource holders. The technique uses nine rules to achieve bounded pi-blocking on cluster-scheduled systems. However, our problem domain differs from that of [4] since: (i) all jobs under consideration are already suspended and (ii) we are "scheduling" positions in queues instead of scheduling actual CPUs. This greatly simplifies the donation process. In addition to these simplifications, donation in the O-KGLP only affects tasks that make use of protected k resources, *so donation is isolated to these participating tasks*. Non-resource-using tasks do not participate and cannot experience pi-blocking as a result. This addresses the limitation discussed earlier in Sec. 1 that arrises when the k-OMLP is used in a globally-scheduled system.

Additional Rules. The following additional rules allow us to maintain Assumption A1.

D1 (Precedes Rule O1.2) If the arrival of R_i in the PQ would cause the eviction from PQ^{HP} of a request U_x, based upon the effective priority of U_x, then the priority of R_i is donated to U_x, R_i is held from entering the PQ, and J_i suspends. Resource holder H_x may transitively inherit the new effective priority of U_x.

D2 R_i ceases to donate its priority to U_x when either

D2.1 U_x enters FQ$_x$, or

D2.2 the arrival of a new request R_h would cause the eviction of U_x with the effective priority of R_i, in which case R_h replaces R_i as a donor to U_x.

D3 R_i enqueues immediately on the PQ after R_i ceases to be a priority donor. This action takes place before the set

[4] One might suggest that requests from PQ^{HP} be dequeued in priority-order to avoid lower-priority requests from preceding R_i. However, doing so can result in an unbounded scenario when $\left|\text{PQ}^{HP}\right| < k$. Suppose there is a single request R_l in PQ^{HP} and resource holder H_x inherits a priority from R_l. After $l^{max} - \varepsilon$ time, a higher-priority request R_h is issued and joins PQ^{HP}; likewise, resource holder H_y inherits a priority from R_h. Soon thereafter, H_x releases its resource, causing R_h to be dequeued from the PQ (priority-order) and moved to FQ$_x$. The progress R_l made has been lost. Further, because R_h arrived after R_l, R_l cannot assume the brief progress made by R_h. This scenario can recurr and R_l makes no progress while in PQ^{HP}.

PQ^{HP} is re-evaluated, since this event may be triggered by a request in PQ^{HP} enqueuing on an FQ.

Let us now show that Assumption A1 holds.

Lemma 6. *U_x is never evicted from PQ^{HP} by the arrival of new, higher-priority, requests in PQ.*

Proof. A request R_d that could cause U_x to be evicted from PQ^{HP} is prevented from entering the PQ while R_d donates its priority to U_x instead. By Rule D1 and D2, R_d is one of the k highest-priority requests in the PQ *and* any donors. Since U_x has the effective priority of R_d, U_x must have one of the k highest effective priorities among requests in *only* the PQ. □

Blocking Analysis Revisited. Jobs that donate their priority experience an additional source of pi-blocking since donor requests are delayed from entering the PQ.

Lemma 7. *A job J_i may experience pi-blocking due to donation bound by*

$$b_i^D = 2 \cdot l^{max}. \tag{4}$$

Proof. Donation may introduce pi-blocking in addition to b^{FQ}, b^{HQ}, and b^{LQ} in two ways: (i) a job may experience pi-blocking while it acts as a donor; and (ii) when its request is delayed by lower-priority requests in PQ^{HP}, which receive a donated priority. Let us first bound the duration of (i).

The donor relationship is established at request initiation, so once a donor ceases to be a donor, it can never be a donor again. Thus, bounding the duration of donation will bound the length of pi-blocking caused by donation.

A donor R_d donates its priority to a donee U_x. By Rule D1, this priority is transitively inherited by resource holder H_x. Thus, H_x makes progress with respect to the priority of R_d; H_x will hold its resource for no longer than l^{max} time while R_d is pi-blocked. Therefore, U_x will be dequeued onto FQ$_x$ in no later than l^{max} time while R_d is pi-blocked, at which point the donor relationship is terminated and R_d joins the PQ.

A request R_i may enter PQ^{LP} while requests with lower base priorities have a higher effective priority, thus leading to the pi-blocking (ii). R_i can be pi-blocked only while its priority is among the top m. Thus, while R_i is pi-blocked as in (ii), each request in PQ^{HP} has an effective priority among the top m, and hence so does each H_x (through inheritance). Thus, R_i can be pi-blocked for a duration of at most l^{max} due to scenario (ii). □

With all the building blocks in place, we may now derive the total pi-blocking a job using the O-KGLP may experience, which is given by

$$b_i = b_i^D + b_i^{LQ} + b_i^{HQ} + b_i^{FQ}. \tag{5}$$

We can now show that the O-KGLP is optimal with O(m/k) pi-blocking.

Theorem 1. *The O-KGLP is optimal with O(m/k) pi-blocking.*

Proof. The maximum pi-blocking, b_i, a job J_i may experience when issuing a request for a resource under the O-KGLP is given by Equation(5). The component terms b_i^D and b_i^{HQ} are both O(1), while the terms b_i^{LQ} and b_i^{FQ} are O(m/k). Thus combined, b_i is O(m/k). This is asymptotically optimal because scenarios can be easily constructed wherein worst-case pi-blocking is $\Omega(m/k)$ under any protocol. □

5 Conclusion

In this paper, we have presented the first real-time k-exclusion locking protocol designed specifically for globally-scheduled JLSP systems. The O-KGLP is asymptotically optimal with respect to the number of system CPUs and also scales inversely with additional resources.

In future work, we will evaluate the performance of the O-KGLP as a part of the schedulability analysis in a larger study that will evaluate various CPU scheduler and GPU locking protocol configurations for use in real-time multiprocessor systems with multiple GPUs.

References

[1] J. Anderson and Y. Kim. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16:2003, 2001.

[2] S. Baruah. Feasibility analysis of preemptive real-time systems upon heterogeneous multiprocessor platforms. In *Proceedings of the 25th IEEE Real-Time Systems Symposium*, pages 37–46, 2004.

[3] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, 2010.

[4] B. Brandenburg and J. Anderson. Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In *Proceedings of the 11th International Conference on Embedded Software*, 2011. To appear.

[5] M.-I. Chen. *Schedulability analysis of resource access control protocols in real-time systems*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[6] G. Elliott and J. Anderson. Globally scheduled real-time systems with GPUs. In *Proceedings of the 18th Real-Time and Network Systems*, 2010.

[7] M. Fischer, N. Lynch, J. Burns, and A. Borodin. Distributed FIFO allocation of identical resources using small shared space. *ACM Trans. Program. Lang. Syst.*, 11(1):90–114, 1989.

[8] P. Gai, L. Abeni, and G. Buttazzo. Multiprocessor DSP scheduling in system-on-a-chip architectures. In *Proceedings of the 14th EuroMicro Conference on Real-Time Systems*, pages 231–238, 2002.

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

[10] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

[11] P. M. Morse. *Queues, Inventories, and Maintenance: The Analysis of Operational System with Variable Demand and Supply*. Wiley, 1958.

[12] R. Pellizzoni and G. Lipari. Holistic analysis of asynchronous real-time transactions with earliest deadline scheduling. *Journal of Computer and System Sciences*, 73:186–206, March 2007.

[13] M. Raynal and D. Beeson. *Algorithms for mutual exclusion*. MIT Press, 1986.