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 growing generality allows for them to be used as powerful co-processors. In previous work, the authors showed that GPUs may be integrated into real-time systems by treating GPUs as shared resources, allocated to real-time tasks through mutual exclusion locking protocols. In this paper, an asymptotically optimal k-exclusion locking protocol is presented for globally-scheduled job-level static-priority (JLSP) systems. This protocol may be used to manage a pool of resources, such as GPUs, 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 This research has found that uniprocessor techniques are often no longer valid or suffer from significant inefficiencies when applied directly to multiprocessor platforms.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 [5]. k-exclusion locking protocols can be used to arbitrate access to pools of similar or identical resources,

[*]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.

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 [9], 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 [9]. 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 paper, we present an asymptotically optimal k-exclusion protocol that can be used to realize this approach in globally-scheduled real-time systems, where optimality is defined in terms of the number of resource requests that may block one another. 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 in sporadic task systems with no utilization loss [14], despite the Dhall effect [8]. 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 have been investigated before. Chen [6] 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.

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 [5].[1] The Clustered k-exclusion OMLP (CK-OMLP) is asymptotically optimal and may be applied to globally-scheduled systems (since a globally-scheduled system is a degenerate case of a

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

cluster-scheduled system where there is only one cluster), but under it, 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, owing to the fact that priorities between clusters are not directly comparable. In order to bound blocking time between clusters, a task may be required to donate its priority to another task instead of being scheduled itself. Hence, any task can experience delays in execution. This behavior is not strictly necessary under global scheduling. However, delays become unbounded, even in the global case, if these behaviors are removed. Delays caused by the CK-OMLP may not be particularly harmful, in terms of schedulability, when the protocol 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 [9]. 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, 17]). 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, 13, 16], 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 different 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 [4], where optimality is defined in terms of the number of resource requests that may block one another. 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 of a task for a shared 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 CK-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 present our k-exclusion locking protocol, the

O-KGLP, and prove that it is asymptotically optimal in the same section. In Sec. 5, we present more detailed analysis to derive tighter blocking bounds to improve schedulability analysis. In Sec. 6, we present results from schedulability experiments, comparing the performance of the O-KGLP to other k-exclusion locking protocols. We end in Sec. 7 with concluding remarks and a discussion of 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 $T^R \subseteq T$ of the tasks requires use of one of the system's k resources. We assume $k \leq m$ since there is little benefit under suspension-oblivious analysis to allowing k simultaneous resource holders when $k > m$ [3].

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 maximum length of time task T_i holds one of the k resources. For tasks $T_i \notin T^R$, $l_i = 0$. The maximum critical section length of any task $T_i \in T$ is denoted by l^{max}. The *relative deadline*, d_i, is the time after which a job is released by when that job should complete. Arbitrary deadlines are supported in this work, *i.e.*, a relative deadline may take any positive value. The *period* of T_i, p_i, is the minimum separation time between job invocations for task T_i. The *utilization* of T_i, u_i, is defined as $u_i \triangleq e_i/p_i$. Task set utilization is defined by $U \triangleq \sum_{T_i \in T} u_i$.

We say that a job J_i is *pending* from the time of its release, a_i, 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. In such a scheduler, the priority of a job is determined at release time and remains fixed until the job completes. Also, because the scheduler is global, jobs may be scheduled on any available system processor. The *global earliest-deadline-first* (G-EDF) scheduling algorithm is one such JSLP global scheduler. Under G-EDF, the job-level static priority of any job is its absolute deadline, $a_i + d_i$, and jobs are prioritized by earliest deadline and are scheduled globally.

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 (as with jobs, we may omit the subscript j if the particular request invocation is inconsequential). 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. We assume that a job may not make nested resource requests, and thus may only hold one of the k resources at a time. Requests that have not yet been allocated a resource are *pending* requests. Resource holding and pending requests together, are *incomplete* requests. Motivated by common GPU usage patterns, we assume that a job requests at most one resource once per job, though the analysis presented in this paper can be generalized to support multiple, non-nested, requests.[2] We let b_i denote an upper bound on the duration of "priority inversion blocking," further defined in Sec. 3, by which 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. The priority with which J_i is scheduled is the *effective priority* of J_i.

[2]Though the asymptotically optimal locking protocol we present in this paper supports multiple, non-nested, requests, we omit this generalization in analysis to prevent notation from becoming too cumbersome. However, please see [3] for a generalized framework that may be applied to the analysis presented here.

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 deadline requirements, are met. In [4], 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 (*i.e.*, tasks that share resources) 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

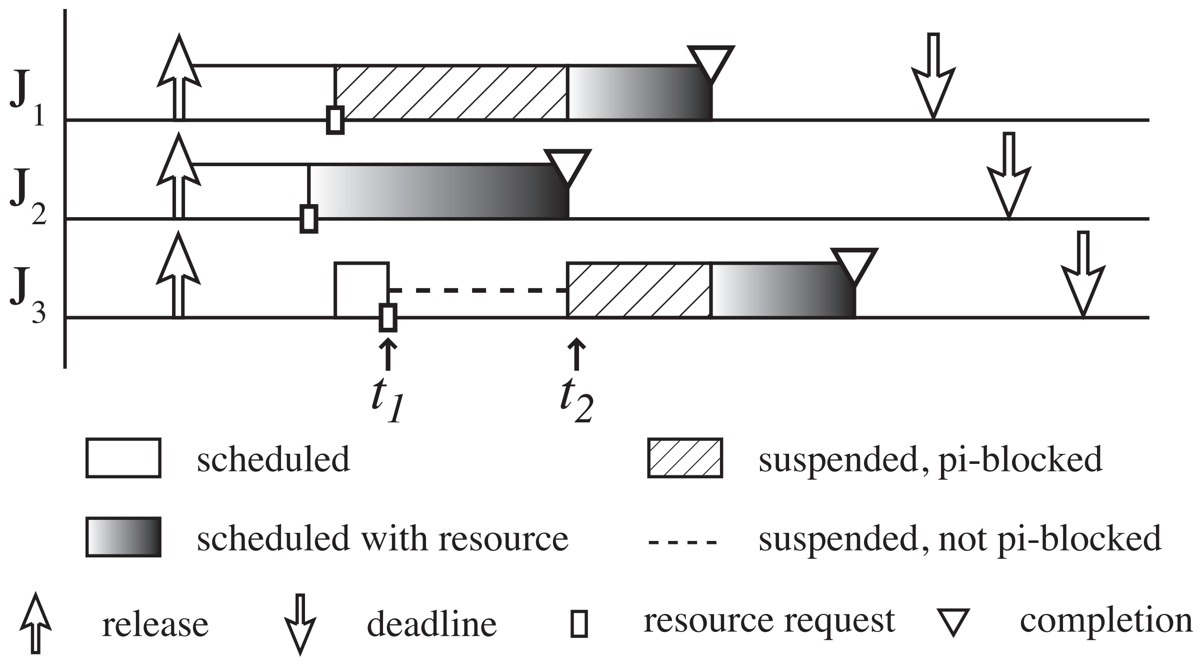

Figure 1: *Job J_3 does not experience pi-blocking within the interval $[t_1, t_2]$ under suspension-oblivious analysis for this two-processor system scheduled by G-EDF. Job J_2 is scheduled within this interval while job J_1 is analytically considered to be scheduled. Job J_3 is not pi-blocked because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

known multiprocessor schedulability analysis techniques for JLSP global schedulers, such as G-EDF, that account for blocking delays are suspension-oblivious.

It was shown in [4] 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 illustrated 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 within 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 [4] that a mutex locking protocol may be considered asymptotically optimal under suspension-oblivious analysis if the maximum number of requests of other tasks that cause pi-blocking is $O(m)$ per resource request—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 scale with the inverse of k (another fixed system parameter). Indeed, the CK-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 CK-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 $O(m/k)$.

By establishing that $\Omega(m/k)$ maximum pi-blocking is sometimes unavoidable in k-exclusion resource arbitration under suspension-oblivious analysis, it follows that any k-exclusion locking protocol with $O(m/k)$ pi-blocking is asymptotically optimal. We can show an $\Omega(m/k)$ pi-blocking lower bound using the same approach taken by Brandenburg *et al.* to establish an $\Omega(m)$ lower bound for mutual exclusion under suspension-oblivious analysis (Lemma 1 in [4]).

Lemma 1. *There exists an arrival sequence for a task set such that, under suspension-oblivious analysis,* $\max_{1 \leq i \leq n}\{b_i\} = \Omega(m/k)$ *under any k-exclusion locking protocol and JLSP scheduler.*

Proof. Without loss of generality, let $T(n)$ denote a task set of n identical tasks that share one pool of k resources protected by an arbitrary k-exclusion lock such that $e_i = 1$, $p_i = 2n$, and $l_i = 1$ for each T_i, where $n \geq m \geq k \geq 2$. Assume that n is an integer multiple of m, and m is an integer multiple of k. Consider the schedule resulting from the following periodic arrival sequence: each $J_{i,j}$ is released at $a_{i,j} = (\lceil i/m \rceil - 1) \cdot m + (j-1) \cdot p_i$, and issues one request $R_{i,j}$ immediately upon release. Jobs are released in groups of m and each job requires one of the k resources protected by the k-exclusion lock for its entire computation. A resulting G-EDF schedule is depicted in Fig. 2.

There is a total of n/m groups of m tasks each. Each group of jobs of tasks $\{T_{g \cdot m+1}, \ldots, T_{g \cdot m+m}\}$, where $g \in \{0, \ldots, n/m - 1\}$, issues m concurrent requests for a resource. Since only k resources may be used simultaneously, any locking protocol must impart some order, and thus there exist k jobs in each group that incur d time units of pi-blocking for each $d \in \{0, \ldots, m/k - 1\}$. Hence, for each g, $\sum_{i=g \cdot m+1}^{g \cdot m+m} b_i \geq k \cdot \sum_{i=0}^{m/k-1} i = \Omega(m^2/k)$, and thus, across all groups,

$$\begin{aligned} \sum_{i=1}^{n} b_i &= \sum_{g=0}^{n/m-1} \sum_{i=g \cdot m+1}^{g \cdot m+m} b_i \\ &= \frac{n}{m} \cdot \Omega\left(\frac{m^2}{k}\right) \\ &= \Omega\left(\frac{nm}{k}\right), \end{aligned} \tag{1}$$

which implies $\max_{1 \leq i \leq n}\{b_i\} = \Omega(m/k)$.

By construction, the schedule does not depend on G-EDF scheduling since no more than m jobs are pending at any time. □ □

Observe that when $k = 1$ (mutual exclusion), the proof for Lemma 1 here matches that of Lemma 1 in [4].

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 that we made did not hold. We will now explain the development process we went through to arrive at an asymptotically optimal k-exclusion

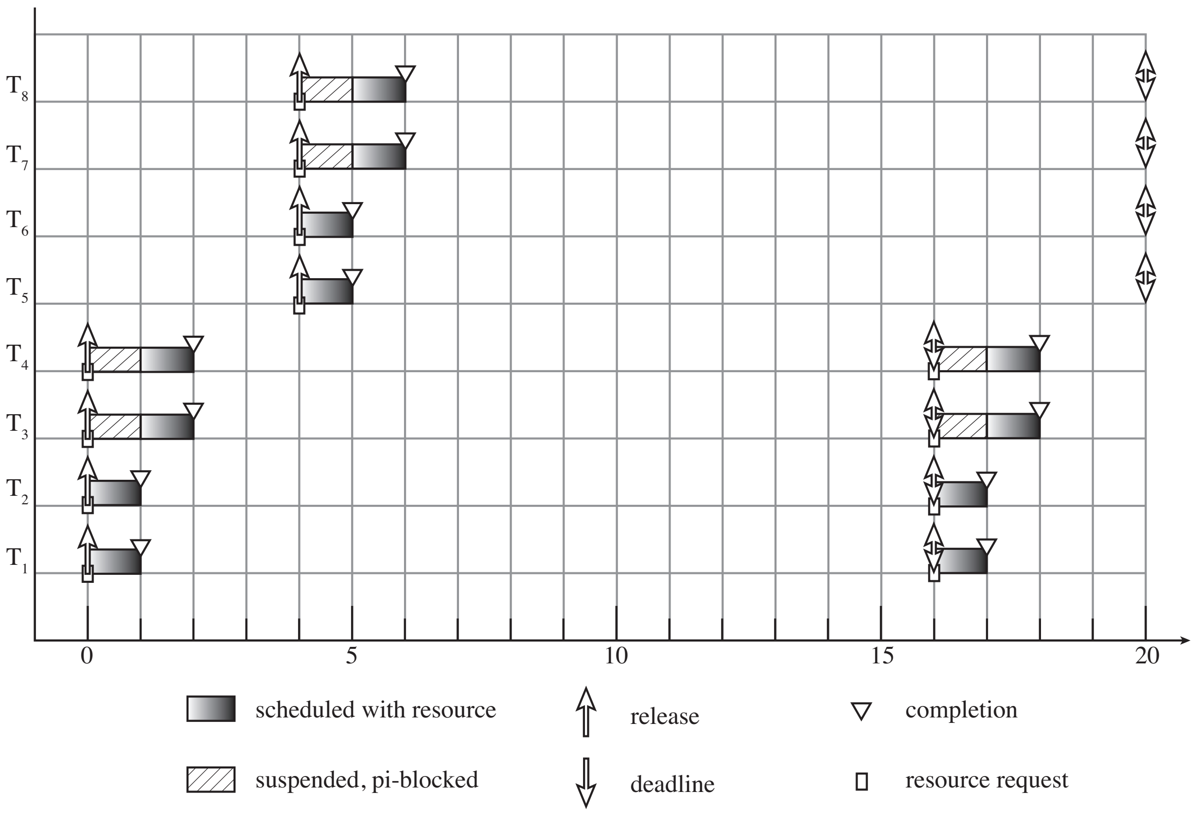

Figure 2: *Depiction of Lemma 1. The illustrated example shows a G-EDF schedule of $T(n)$ for $n = 8$, $m = 4$, and $k = 2$, and thus $g \in \{0, 1\}$. Jobs request a resource immediately upon release. The first group of jobs is released at time 0; the second group is released at time 4. Each group incurs $0 + 0 + 1 + 1 = k \cdot \sum_{i=0}^{m/k-1} i$ total suspension-oblivious pi-blocking.*

locking protocol, which we present thereafter.

4.1 A Single Queue

A classic result from queuing theory states that a single wait-queue is the most efficient method for ordering resource requests for a pool of resources [15]. 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 [12] (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 a distinct request in 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

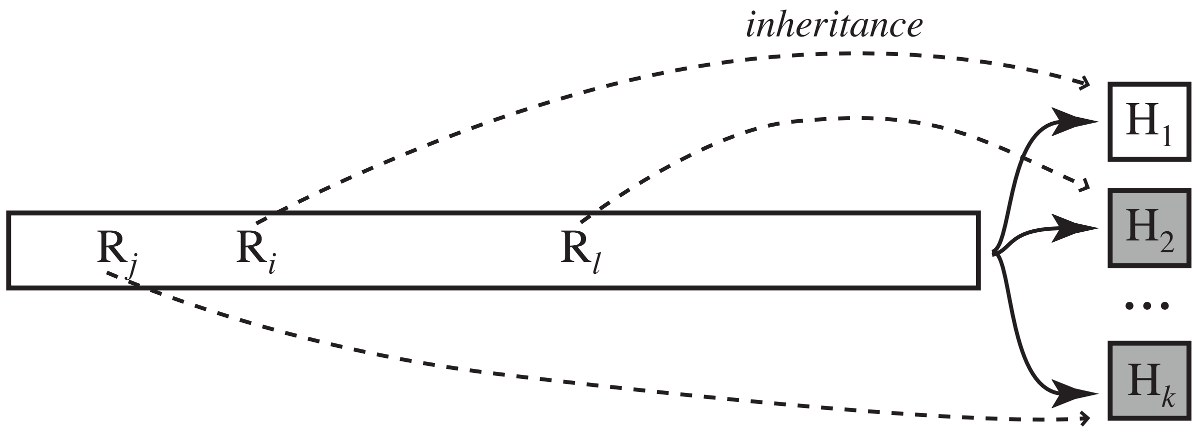

Figure 3: *Pending requests ahead of R_i may be serialized through a single resource when a single wait-queue is used. Depicted above, R_i is the highest-priority request and resource holder H_1 inherits R_i's priority. The remaining $k-1$ resource holders inherit priority from pending requests with priorities less than R_i's. 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.*

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. 3, where R_i may be pi-blocking by $O(n-k)$ other requests. With a little work, it is possible to combine methods from the Bank Algorithm and the OMLP to arrive at an $O(m-k)$ locking protocol. However, this still falls short of our desired optimal bound of $O(m/k)$.

In a non-real-time context, it is often implicitly assumed that all resource holders execute simultaneously. However, this guarantee cannot be maintained in a real-time system with $n > m$ since the priority of the highest-priority job can only be inherited by a single resource holder.[3] It appears that for traditional sporadic real-time systems, a single queue approach will not yield an asymptotically 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.

[3]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.

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 the pi-blocking under these techniques is a function of fixed system parameters instead of task set size, this pi-blocking still does not meet our definition of optimality under k-exclusion and do 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 $\lceil m/k \rceil$, are assigned to each of the k resources (without loss of generality, we assume k evenly divides m for the remainder of this section). An additional 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. Fig. 4 depicts the queue structure of the O-KGLP and inheritance relations derived from the following rules.

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.

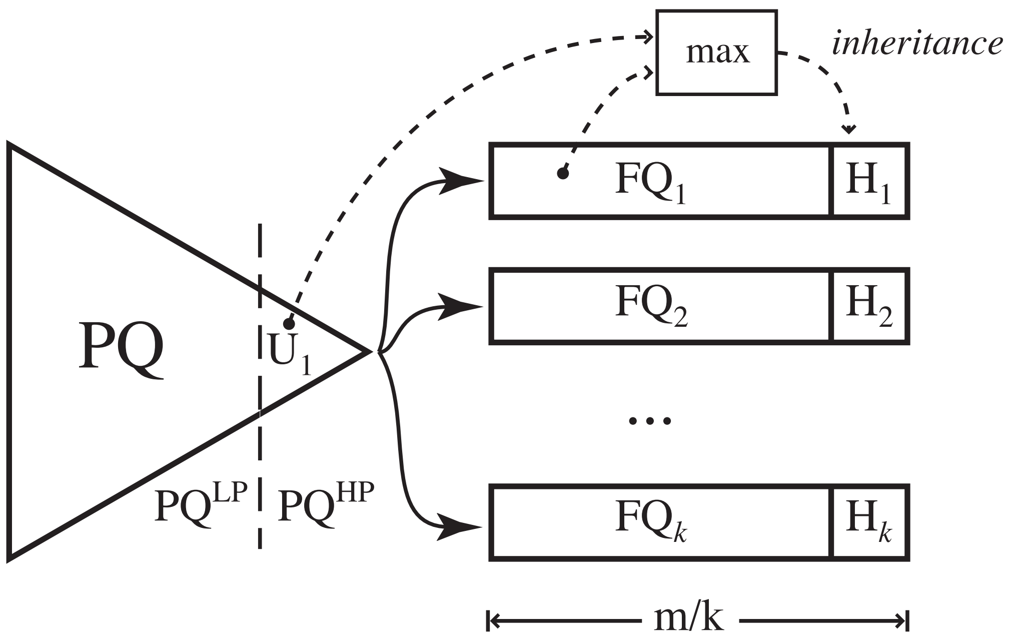

Figure 4: *Queue structure and priority inheritance relations used by the O-KGLP.*

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 distinct request among the k highest-priority pending requests in the PQ, whichever has greater priority. Each resource holder claims a distinct 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.[4] Further, the claimed distinct request (if it exists) from among 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 distinct (unique) request in PQ^{HP} associated with H_x by Rule O3; note that no two resource holders

[4]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 practice, but does not improve upon the worst case.

may claim the same request in PQ^{HP}. Finally, let PQ^{LP} (for "low priority") denote the set of requests in PQ that are not in PQ^{HP}.

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.

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 that 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 units 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 time 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:

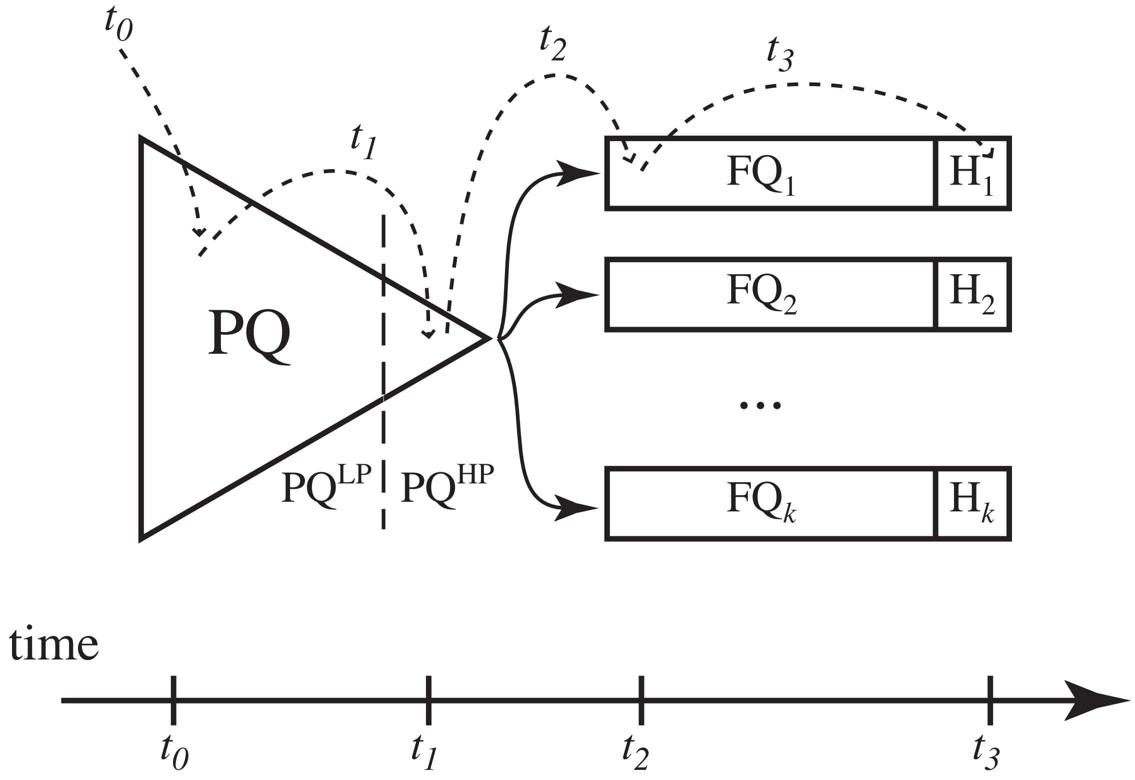

Figure 5: *Job J_i may experience pi-blocking in three time intervals: first in the interval $[t_0, t_1)$, from when J_i's request, R_i, enters the PQ to when the request joins the set PQ^{HP}; next, in the interval $[t_1, t_2)$, which is the duration R_i is in PQ^{HP}; and finally, in the interval $[t_2, t_3)$, which is the time R_i must wait in an FQ until it receives a resource.*

$b_i = b_i^{LQ} + b_i^{HQ} + b_i^{FQ}$. These phases are depicted in Fig. 5.

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 $\left|T^R\right| \leq k$, then no job is ever pi-blocked ($b_i = 0$) since every request can be trivially satisfied simultaneously. If $k < \left|T^R\right| \leq m$, 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 if $m < \left|T^R\right| \leq m + k$. A job can only experiences pi-blocking in every phase if $\left|T^R\right| > m + k$. 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.

Lemma 2. *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's 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 $k < \left|T^R\right| \leq m$, 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 $\left\lfloor \frac{|T^R \backslash \{T_i\}|}{k} \right\rfloor$ in length since the length of any FQ may only deviate from the average FQ length by more than one. Thus, $\left\lfloor \frac{|T^R|-1}{k} \right\rfloor$ upper-bounds the number of lower-priority jobs that may pi-block J_i when $k < \left|T^R\right| \leq m$. □ □

Lemma 3. *J_i experiences pi-blocking while R_i is queued on* FQ$_x$ *of at most*

$$b_i^{FQ} = \min\left(\frac{m}{k} - 1, \left\lfloor \frac{\left|T^R\right| - 1}{k} \right\rfloor\right) \cdot l^{max}. \tag{2}$$

Proof. J_i experiences worst-case pi-blocking when the jobs that pi-block it have the longest possible critical sections. By Lemma 2 and by upper-bounding critical section lengths with l^{max}, the proof follows. □ □

Lemma 4. *J_i may only be pi-blocked for the duration of one critical section while its request is in the set PQ^{HP}. Thus,*

$$b_i^{HQ} = l^{max} \tag{3}$$

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 units 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 5. *Progress is guaranteed for any request, R_i, pending in PQ^{LP}.*

Proof. Assumption A1 ensures that each $U_x \in$ PQ^{HP} has a priority no less than that of $R_i \in$ PQ^{LP}. Thus, by Rule O3, each H_x is scheduled with an effective priority greater than R_i while $R_i \in$ 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 is yet to be determined. □ □

Lemma 6. *Job J_i, with request $R_i \in$ PQ^{LP}, is pi-blocked for at most $\frac{m}{k} \cdot l^{max}$ time. Thus,*

$$b_i^{LQ} = \frac{m}{k} \cdot l^{max}. \tag{4}$$

Proof. A job is not pi-blocked under suspension-oblivious analysis when there exist at least m other pending higher-priority jobs. By Lemma 5, all the resource holders in the FQs are scheduled with an effective priority at least that of R_i while $R_i \in$ PQ^{LP}. Consequently, all potential lower-priority requests in the FQs when J_i issued R_i at time t_0 (see Fig. 5) 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 time $t_0 + \frac{m}{k} \cdot l^{max}$ (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

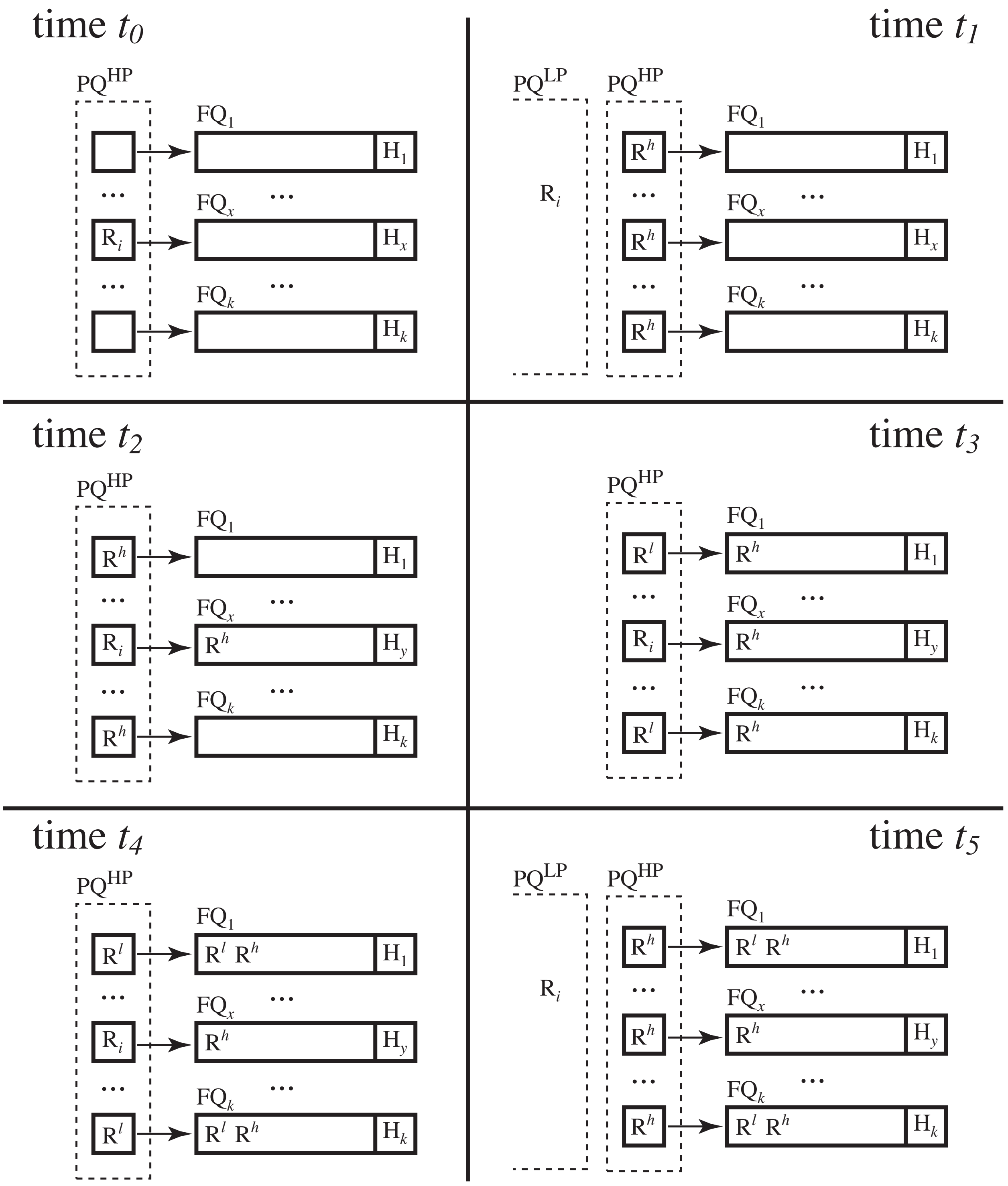

Figure 6: *Unbounded pi-blocking for b_i^{HQ} if evictions from PQ^{HP} are allowed.* **t_0**: *R_i is in PQ^{HP} and H_x inherits the priority of R_i.* **t_1**: *R_i is evicted from PQ^{HP} by the arrival of k higher-priority requests.* **t_2**: *Resource holder H_y completes; R_i rejoins PQ^{HP}.* **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}.*

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. 6.

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 its eviction from PQ^{HP} has been lost.

Still, perhaps the *number* of times R_i can be evicted, *while* R_i remains pi-blocked, can be 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 PQ$^{HP}\setminus\{R_i\}$ (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 to end pi-blocking under suspension-oblivious analysis.

It could be reasonably suggested that requests from PQ^{HP} should be dequeued in priority-order to avoid lower-priority requests from preceding R_i. However, doing so can result in an unbounded scenario when $\left|\text{PQ}^{HP}\right| < k$. Such a scenario is illustrated in Fig. 7. Suppose there is a single request R^l in PQ^{HP} and resource holder H_x inherits R^l's priority at time t_0.

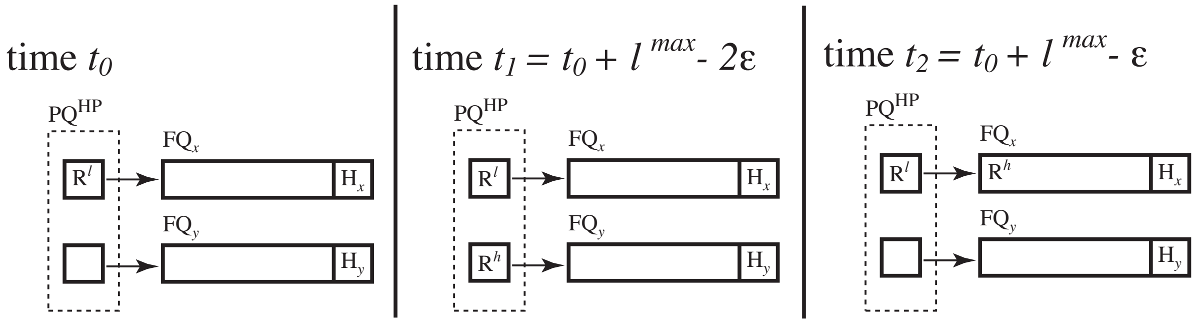

Figure 7: *Dequeueing requests from PQ^{HP} in priority-order (in violation of Rule O4) can lead to starvation. At time t_0, H_x inherits priority from R^l. At time t_1, R^l has made $l^{max} - 2\varepsilon$ time units of progress. Also at time t_1, a higher priority request R^h enters PQ^{HP}. At time t_2, H_x completes and R^h is dequeued from the PQ (priority-order) and moved to FQ_x. The progress made by R^l (now $l^{max} - \varepsilon$ time units) has been lost. Progress made by R^h (ε time units) could be given to R^l, but it is not enough to compensate for the lost progress. This scenario may repeat indefinitely, resulting in the starvation of R^l.*

Further suppose at time $t_1 = t_0 + l^{max} - 2\varepsilon$, R^l has made $l^{max} - 2\varepsilon$ time units of progress. Also at time t_1, a higher-priority request R^h is issued and joins PQ^{HP}, and resource holder H_y inherits R^h's priority. Soon thereafter, at time $t_2 = t_0 + l^{max} - \varepsilon$, H_x releases its resource, causing R^h to be dequeued from the PQ (priority-order) and moved to FQ$_x$. The progress made by R^l (now $l^{max} - \varepsilon$ time units) has been lost. Further, the brief progress R^h made while in PQ^{HP}, $t_2 - t_1 = \varepsilon$ time units, cannot be transferred to R^l to fully compensate R^l because R^h arrived after R^l.[5] This scenario can recur and R^l starves in PQ^{HP}.

Merely disallowing PQ^{HP} evictions will not resolve these issues since doing so trades one k-exclusion problem (resources) for another (the k positions in PQ^{HP}). 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.* [5], priority donation is priority inheritance technique that allows for the bounding of pi-blocking on cluster-scheduled systems. Under priority donation, jobs with higher priorities may temporarily suspend and donate their priority to

[5]This transferal of progress could be made by forcing H_y to claim R^l from PQ^{HP} (Rule O3).

resource holders. The technique uses nine rules to achieve bounded pi-blocking on cluster-scheduled systems. However, our problem domain differs from that of [5] 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 CK-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 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 7. *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 Rules 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 8. *A job J_i may experience pi-blocking due to donation of at most*

$$b_i^D = 2 \cdot l^{max}. \tag{5}$$

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}, that have received 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, and thus, R_d makes progress. H_x will hold its resource for no longer than l^{max} time (with respect to the priority of R_d) 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 as in (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 since R_i is among the top m and is not a donor

(k donors must exist with priorities greater than R_i's), and hence, through inheritance, H_x also has a priority among the top m. Thus, R_i can be pi-blocked for a duration of at most l^{max} due to scenario (ii): H_x will complete within l^{max} time units. □ □

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

$$\begin{aligned} b_i &= b_i^D + b_i^{LQ} + b_i^{HQ} + b_i^{FQ}. \\ &= \left(\tfrac{m}{k} + \min\left(\tfrac{m}{k} - 1, \left\lfloor \tfrac{|T^R|-1}{k} \right\rfloor\right) + 3\right) \cdot l^{max} \\ &\leq 2\left(\tfrac{m}{k} + 1\right) \cdot l^{max} \end{aligned} \tag{6}$$

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

Theorem 1. *The O-KGLP is asymptotically 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 (6). 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, b_i is O(m/k). This is asymptotically optimal because worst-case pi-blocking is $\Omega(m/k)$ under any protocol, as shown by Lemma 1. □ □

Supporting $k > m$. In Sec. 2, it was assumed $k \leq m$ since there is little benefit under suspension-oblivious analysis to allowing k simultaneous resource holders when $k > m$ [3]. However, it may be beneficial to allow $k > m$ in cases where resources holders may self-suspend, as can happen when GPUs are used to perform computations. When a resource holder self-suspends, the $(m+1)^{st}$ highest-priority resource holder may then have an opportunity to execute. This can improve average case performance.

Minor modifications to the O-KGLP may be made to support instances where $k > m$. Specifically, Rule O1.1 must be revised so that R_i enqueues on the shortest FQ$_x$ if $queued(t_0) < \mathbf{max}(m, \boldsymbol{k})$. Also, the size of PQ^{HP} must be redefined to be $\min(\boldsymbol{m}, k, |PQ|)$.

The above proofs go through with minor revisions to accommodate these changed parameters. However, despite the possibility of improved performance in practice, the worst-case bounds on blocking are no better than the case when $k = m$. Thus, even in cases where resource holders may self-suspend, it may be better to statically allocate each "extra" resource to one resource-using task instead of using resource pools with $k > m$. Alternatively, resources could be partitioned into clusters of no more than m in size and each cluster protected by a distinct O-KGLP lock. Resource-using tasks would be partitioned onto these clusters. Either approach would result in schedulability at least as good as, but likely better than, using the O-KGLP with $k > m$.

5 Detailed Blocking Analysis

Equation (6) gives an upper bound on pi-blocking that any resource-using job may experience. While this bound is asymptotically optimal, it is still pessimistic since all critical section lengths are assumed to be l^{max} in length. In practice, the critical section lengths of resource-using tasks may vary. In this case, tighter bounds on pi-blocking can be derived, as we show next.

The maximum pi-blocking that a request R_i may experience under the O-KGLP is partly dependent upon the maximum number of incomplete requests that may exist when R_i is issued. Due to intra-task job precedence constraints, there may exist at most one incomplete resource request per task in T^R at any given instant. Thus, the number of requests that may block (or "interfere" with) R_i is $|T^R \backslash \{T_i\}| = |T^R| - 1$. This number of requests determines whether J_i experiences pi-blocking in each of the phases (that is, b^D, b^{LQ}, b^{HQ}, b^{FQ}) of the O-KGLP. For example, if $|T^R| \leq k$, then a request R_i can always be immediately granted and no tasks experience blocking (we will not revisit analysis for this trivial case). Let us next consider the case when $k < |T^R| \leq m$.

Definition 1. *Let the function $top(v, \mathcal{S})$ denote the v longest requests in the set of requests $\mathcal{S}$, where request length is given by $|R_i| \triangleq l_i$.*

Definition 2. *Let the maximum set of requests that may exist when R_i is issued be denoted by $\mathcal{I}_i \triangleq \{R_j \mid T_j \in T^R \backslash \{T_i\}\}$.*

Theorem 2. *When $k < |T^R| \leq m + k$, the maximum pi-blocking experienced by job J_i with request R_i is*

$$b_i = \sum_{R_j \in top\left(\left\lfloor \frac{|T^R|-1}{k} \right\rfloor, \mathcal{I}_i\right)} |R_j|. \tag{7}$$

Proof. Let us first consider the case where $k < |T^R| \leq m$ holds. Recall from Lemma 2 that a job J_i may be pi-blocked by at most $\lfloor(\left|T^R\right| - 1)/k\rfloor$ requests when $|T^R| \leq m$. Thus, Theorem 2 follows in this case from Lemma 2 and intra-task precedence constraints.

We now consider the remaining case where $m < |T^R| \leq m + k$ holds. The FQs and PQ^{HP} can hold up to $m + k$ requests, which is at most the maximum number of incomplete requests when $|T^R| \leq m + k$. Thus, no job may act as priority donor since no request can enter PQ^{LP}. Assumption A1 trivially holds as a result, so no request may precede R_i into the FQ to which R_i is bound while it is in PQ^{HP} (Rule O3). Thus, $\mathcal{I}_i$ is the set of requests that may interfere with R_i. By Rules O3 and O4, R_i is blocked by at most $\lceil m/k \rceil$ other requests (one while R_i is in PQ^{HP} and $\lceil m/k \rceil - 1$ while R_i is in an FQ). When $|T^R| = m+k$, $\lfloor(\left|T^R\right|-1)/k\rfloor = \lfloor(m+k-1)/k\rfloor = \lfloor(m-1)/k\rfloor+1 = \lceil m/k \rceil$, which is equal to the maximum number of requests that may block R_i. Thus, Theorem 2 follows. □ □

This completes the detailed blocking analysis of the O-KGLP when $\left|T^R\right| \leq m + k$. We now consider remaining case where $\left|T^R\right| > m + k$ holds. This case is more complex than the previous cases since requests may be enqueued in PQ^{LP}. This allows requests issued after R_i to advance ahead of R_i in the queues; this even includes multiple requests from unique jobs of the same task. Thus, $\mathcal{I}_i$ no longer captures all of the requests that may interfere with R_i. A new interference set, $\mathcal{S}_i$, must be derived.

A set of consecutive *generic requests* (*i.e.*, virtual requests defined for analysis purposes) of a task T_j that may interfere, in the worst-case, with a single request R_i from task T_i can be computed for hard real-time and soft real-time (where soft real-time is defined by bounded deadline tardiness) globally-scheduled systems using formulas from [4] and [9], respectively.[6] This set is denoted by $\mathit{tif}(T_i, T_j)$.

Definition 3. *For hard real-time systems,*

$$\mathit{tif}(T_i, T_j) \triangleq \left\{ R_{j,v} \mid 1 \leq v \leq \left\lceil \frac{p_i + rt_j}{p_j} \right\rceil \right\}, \tag{8}$$

where rt_j is the worst-case response time of a job of T_j [4].

Definition 4. *For soft real-time systems (under the "bounded tardiness" definition of soft real-time),*

$$\mathit{tif}(T_i, T_j) \triangleq \left\{ R_{j,v} \mid 1 \leq v \leq \left\lceil \frac{p_i + x_i + p_j + x_j}{p_j} \right\rceil \right\}, \tag{9}$$

where x_i and x_j denote tardiness bounds of job completion for tasks T_i and T_j, respectively [9].

We say that $\mathit{tif}(T_i, T_j)$ defines a set of generic requests because request $R_{j,v} \in \mathit{tif}(T_i, T_j)$ does not denote the v^{th} request made by task T_j after the release of T_j's first job. Rather, $R_{j,v}$ denotes the v^{th} resource request in a worst-case string of consecutive requests of T_j that may interfere with request R_i of T_i.

To gain an intuitive understanding of $\mathit{tif}(T_i, T_j)$, let us consider the upper limit of v.[7] The upper limit of v denotes the maximum number of jobs of T_j that may execute over the time interval during which the request R_i may be made. We assume that R_i may be issued at any time while J_i is executing, so this interval is bounded by p_i and $p_i + x_i$ in the hard and soft real-time cases, respectively. We divide this interval by the minimum separation time

[6]The formulas of [4] and [9] support multiple, non-nested, requests to be made. As mentioned in Sec. 2, the O-KGLP supports multiple, non-nested, requests though we omit this generalization in analysis to prevent notation from becoming too cumbersome.

[7]Please refer to [4] and [9] for proofs of Equations (8) and (9), respectively.

between jobs of T_j, p_j. We must also incorporate a carry-in job for T_j that may be released prior J_i. This entails increasing the interval under consideration by the worst-case response time of job J_j, which is rt_j and $p_j + x_j$ in the hard and soft real-time case, respectively. Thus, $\mathit{tif}(T_i, T_j)$ generates the set of requests from T_j that may interfere with R_i.

We aggregate the interference from each task to derive $\mathcal{S}_i$ (using the hard and soft real-time variants as appropriate):

Definition 5.

$$\mathcal{S}_i \triangleq \bigcup_{T_j \in T^R \setminus \{T_i\}} \mathit{tif}(T_i, T_j). \tag{10}$$

Equation (10) gives us the set of requests that may interfere with R_i, but not all requests may cause pi-blocking. An upper bound on the number of interfering requests can be derived using the same steps to derive Equation (6).

Lemma 9. *When $\left|T^R\right| > m + k$, the maximum number of requests that pi-block J_i is*

$$2\left(\left\lceil \frac{m}{k} \right\rceil + 1\right). \tag{11}$$

Proof. Follows from Lemmas 2, 4, 6, and 8. In other words, this is Equation (6), with l^{max} removed. (Note that the term $\lfloor (|T^R| - 1)/k \rfloor$ drops out of Equation (6) when $\left|T^R\right| > m + k$.) □ □

Computing the worst-case pi-blocking experienced by J_i when $\left|T^R\right| > m + k$ is straightforward using Equations (10) and (11).

Theorem 3. *When $\left|T^R\right| > m + k$, the maximum pi-blocking experienced by J_i is*

$$b_i = \sum_{R_j \in \mathit{top}(2(\lceil m/k \rceil + 1),\ \mathcal{S}_i)} |R_j|. \tag{12}$$

Proof. Follows from Lemma 9. □ □

While the O-KGLP is asymptotically optimal and the above analysis offers tighter bounds than Equation (6), there are cases where an $\mathrm{O}(n/k)$ protocol can offer better schedulability when $\left|T^R\right|$ is sufficiently small. One such $\mathrm{O}(n/k)$ protocol is the k-FMLP [10]. When a resource request is issued under the k-FMLP, the request is enqueued on the shortest of k FIFO queues, one per k resource. The k-FMLP resembles the O-KGLP without restrictions on FQ length (so there is no PQ or donors, as well). As a result, Equation (7) may be used to compute the worst-case pi-blocking of a request under the k-FMLP for task sets of any size. Note that this also implies that non-resource-using tasks do not experience pi-blocking.

The k-FMLP offers better schedulability than the O-KGLP when a greater number of requests may block R_i under the O-KGLP than under the k-FMLP. Specifically, when $2(\lceil m/k\rceil + 1) > \lfloor(\left|T^R\right| - 1)/k\rfloor$.[8] For example, suppose $\left|T^R\right| = m + k + 1$. Under the O-KGLP, b_i is composed of $2(\lceil m/k\rceil + 1)$ critical section lengths from interfering requests. However, under the k-FMLP, b_i is composed of only $\lfloor(\left|T^R\right| - 1)/k\rfloor = \lfloor(m + k)/k\rfloor$ critical section lengths from interfering requests. This is at least half as many as the O-KGLP.

For a *specific* task set under which the k-FMLP yields better schedulability than the standard O-KGLP, one may opt to use the k-FMLP instead. Alternatively, one may also increase the length of the O-KGLP FQs to $\lceil\left|T^R\right|/k\rceil$ and allow up to a total of $\left|T^R\right|$ requests to be enqueued in the FQs. This essentially deactivates the PQ and donor rules, and the O-KGLP operates as if it were the k-FMLP. This "enhanced" O-KGLP may be favorable to the k-FMLP from a system designer's perspective since only a single locking protocol implementation is required, provided that the FQs can be resized when appropriate.[9] The O-KGLP, with this task-set-specific enhancement, is guaranteed to perform at least as well as the k-FMLP, and due to asymptotic optimality, will outperform the k-FMLP by larger

[8]There are additional cases when the k-FMLP may still offer better schedulability than the O-KGLP, even if $2(\lceil m/k\rceil + 1) > \lfloor(\left|T^R\right| - 1)/k\rfloor$ does not hold. This is because a job J_i with a single request can be pi-blocked by requests of multiple jobs of task T_j under the O-KGLP. Jobs of T_j may have long critical sections in comparison to other tasks, displacing shorter requests in the set of requests determined by $top(v, S)$. In contrast, only one critical section per task is considered under the k-FMLP.

[9]This is trivial if FQs are implemented with linked lists.

Protocol	Utilization	Schedulable?	b_{using}	$b_{non\text{-}using}$
k-FMLP	4.25	no	3.5	0.0
CK-OMLP	4.75	no	1.5	1.0
O-KGLP	4.0	yes	3.0	0.0

Table 1: *Soft real-time schedulability for an example task set under the k-FMLP, CK-OMLP, and O-KGLP on a system with $m = 4$ and $k = 2$.*

margins as n grows.

Example. We now present comparative schedulability analysis of the k-FMLP, CK-OMLP, and O-KGLP for an example task set to illustrate the advantages of the O-KGLP over previous protocols. We want to schedule a task set of resource-using and non-resource using tasks with G-EDF for a soft real-time system where deadline tardiness must be bounded. Tardiness is bounded, *i.e.*, the system is soft real-time schedulable, if the total task set utilization (with blocking time treated as execution time) is no greater than m [7], provided no task has an individual utilization greater than one.

Suppose we have a system consisting of four CPUs ($m = 4$) and a resource pool size of two ($k = 2$). Our task set consists of 30 tasks; half of the tasks are resource-using with a period of $p_{using} = 30$, execution time of $e_{using} = 2$, and critical section length of $l_{using} = 0.5$; the remaining tasks are non-resource-using with a period of $p_{non\text{-}using} = 10$ and execution time of $e_{non\text{-}using} = 1$. Blocking terms for each protocol are calculated using the algorithms given in Appendix A.

Table 1 presents schedulability analysis results. Blocking time for resource-using tasks is the greatest under the k-FMLP, with $b_{using} = 3.5$. Under the O-KGLP it is less: $b_{using} = 3.0$. Blocking time for resource-using tasks is the least under the CK-OMLP, with $b_{using} = 1.5$. While non-resource-using tasks experience *no* blocking under the k-FMLP and O-KGLP, these tasks experience blocking of $b_{non\text{-}using} = 1.0$ under the CK-OMLP. However, with all blocking terms considered, the task set is only schedulable under the O-KGLP since total task set utilization is 4.0.

6 Schedulability Experiments

To better understand the schedulability properties of the O-KGLP, we randomly generated task sets with varying characteristics. Soft real-time schedulability under global earliest-deadline-first scheduling was determined, as described in [11] for tasks with relative deadlines equal to periods ($d_i = p_i$), using the detailed blocking analysis presented in Sec. 5. We focus our attention on soft real-time schedulability since global schedulers (the only type the O-KGLP supports) are capable of ensuring bounded deadline tardiness in sporadic task systems with no utilization loss [14]. Schedulability was also tested under different locking protocols for comparison. These were Brandenburg *et al.*'s clustered CK-OMLP, mentioned in Sec. 4, and the k-FMLP and enhanced O-KGLP, discussed in Sec. 5.

Experimental Setup. The task set characteristics varied by per-task utilization, resource pool size (k), critical section length, and number of resource-using tasks in a task set. *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 *pool size*, $k \in \{2, 4, 6, 8\}$, determines the number of times a resource was replicated in each scenario. *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 [9], while very short and short intervals may be common to other shared resources. *Percentage number 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 experimental scenario was defined by any permutation of these four parameters for an eight CPU system, yielding a total of 480 scenarios.

We generated random task sets for each 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 selected tasks for T^R from the task set; we determined the number of resource-using tasks by selecting a percentage from the percentage number 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 for each tested locking protocol. As per suspension-oblivious analysis, task execution times were inflated by its bound on pi-blocking ($i.e.$, $e_i^{inflated} = e_i + b_i$), prior to performing the soft real-time schedulability test. We tested a total of 5,000,000 task sets for each scenario.

Results. A selection of results that demonstrate observable trends across all scenarios is presented here.

Observation 1. *The O-KGLP can schedule more task sets than the k-FMLP when there are a large number of resource-using tasks.* The $O(n/k)$ pi-blocking experienced under the k-FMLP grows with task set size. In contrast, the asymptotic optimality of the O-KGLP becomes evident when there are a large number of resource-using tasks. This can be seen in Fig. 8, where the small per-task utilizations lead to task sets made up of many light weight tasks. However, as the number of resource-using tasks decreases, schedulability under the standard O-KGLP matches, or performs marginally worse, than the k-FMLP for the reasons discussed at the end of Sec. 5. This is observable in Fig. 9.

Observation 2. *The enhanced O-KGLP offers better schedulability than both the k-FMLP and standard O-KGLP.* As expected by construction, schedulability is better under the enhanced O-KGLP than either the k-FMLP or standard O-KGLP. Schedulability under the enhanced O-KGLP tightly follows that of the k-FMLP in Fig. 9 and the standard O-KGLP in

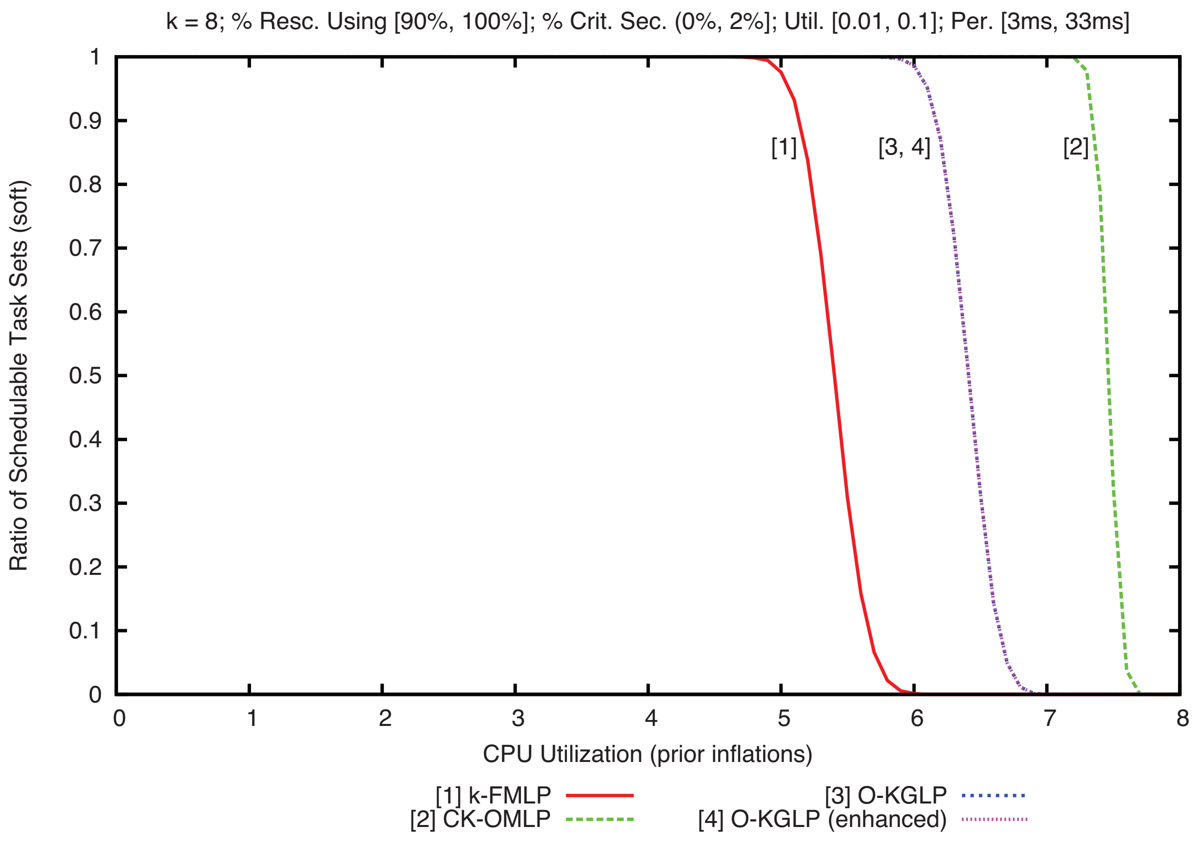

Figure 8: *Schedulability under the O-KGLP is better than the k-FMLP when there are many resource-using tasks. This is the case in this scenario, where task sets are made up of many tasks (since per-task utilization is light) and the percentage of resource-using tasks is high (over 90%). However, the CK-OMLP does perform better than the O-KGLP in this scenario since there are few non-resource using tasks.*

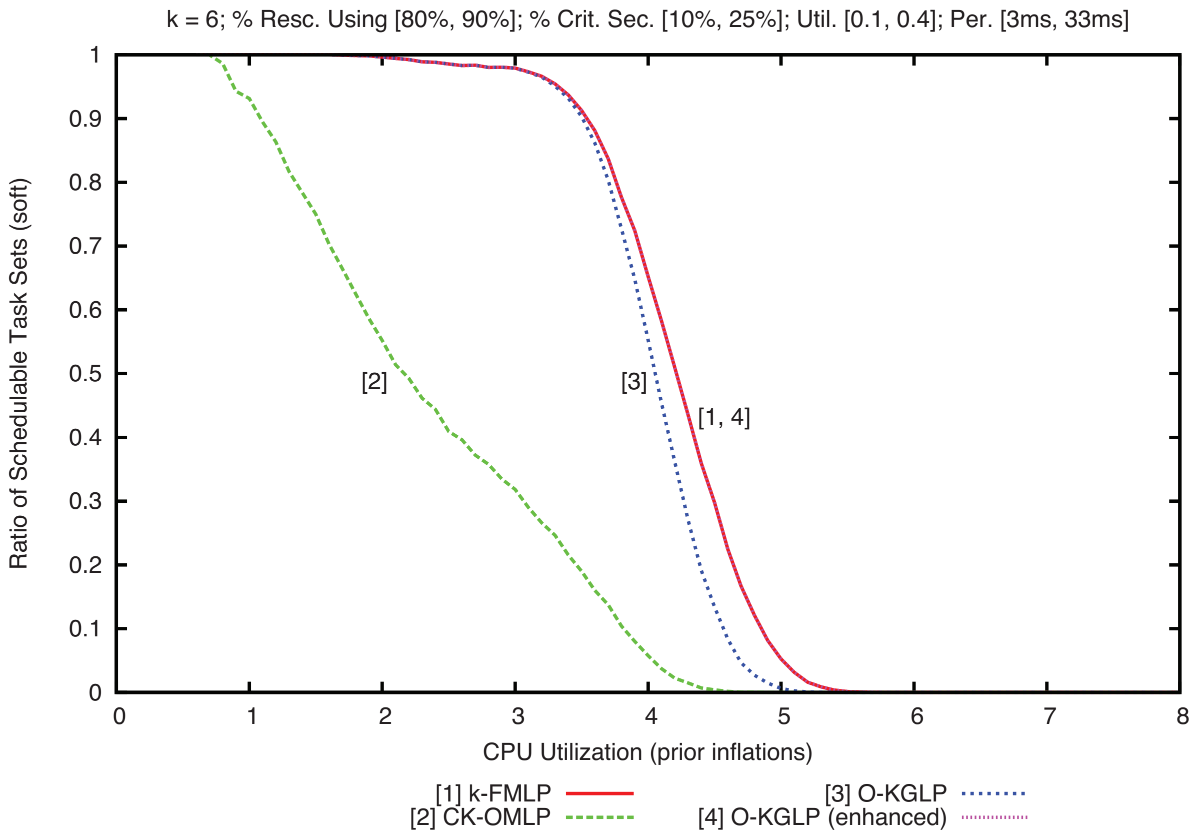

Figure 9: *Schedulability under the k-FMLP can be better than the O-KGLP when there are fewer resource-using tasks, as is the case in this scenario with medium per-task utilizations. Schedulability under the CK-OMLP is poor in comparison to both the k-FMLP and O-KGLP since non-resource-using tasks experience pi-blocking, despite the fact that non-resource-using tasks make up no more than 20% of each task set.*

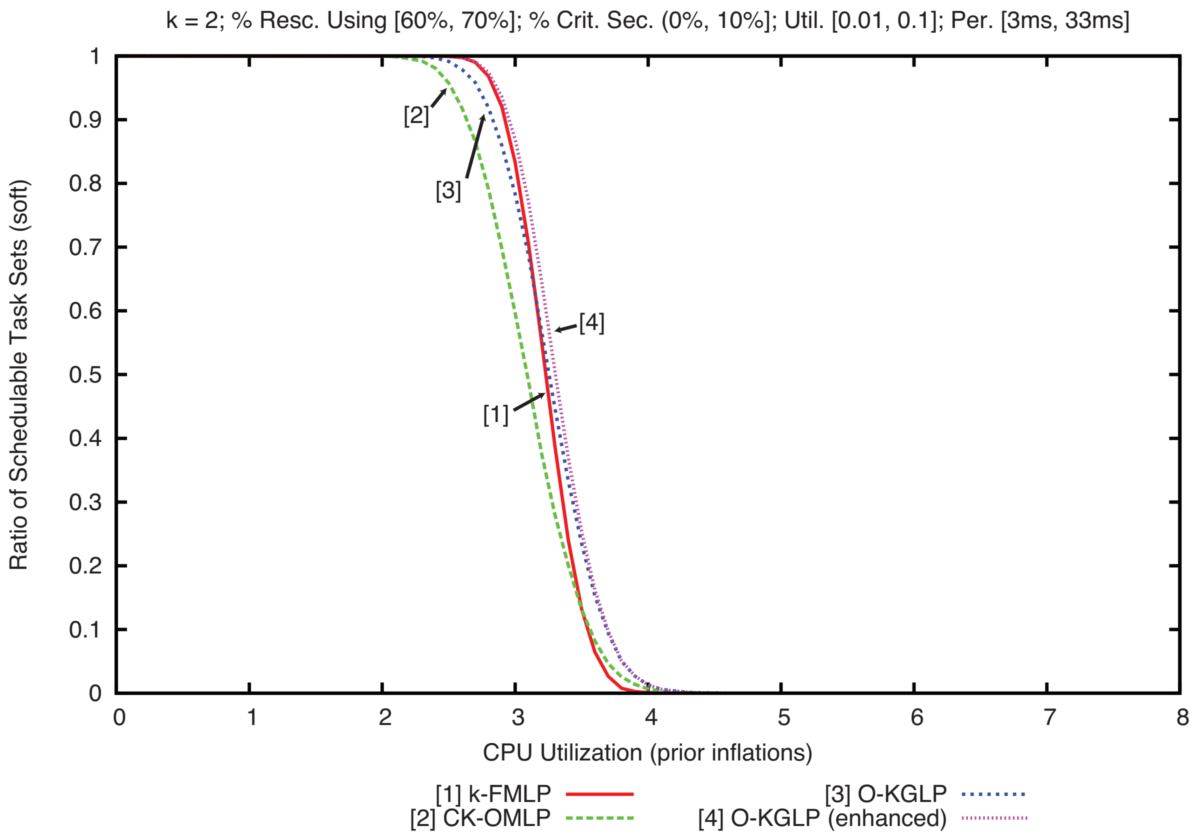

Figure 10: *The enhanced O-KGLP may offer better schedulability than exclusively the k-FMLP or standard O-KGLP.*

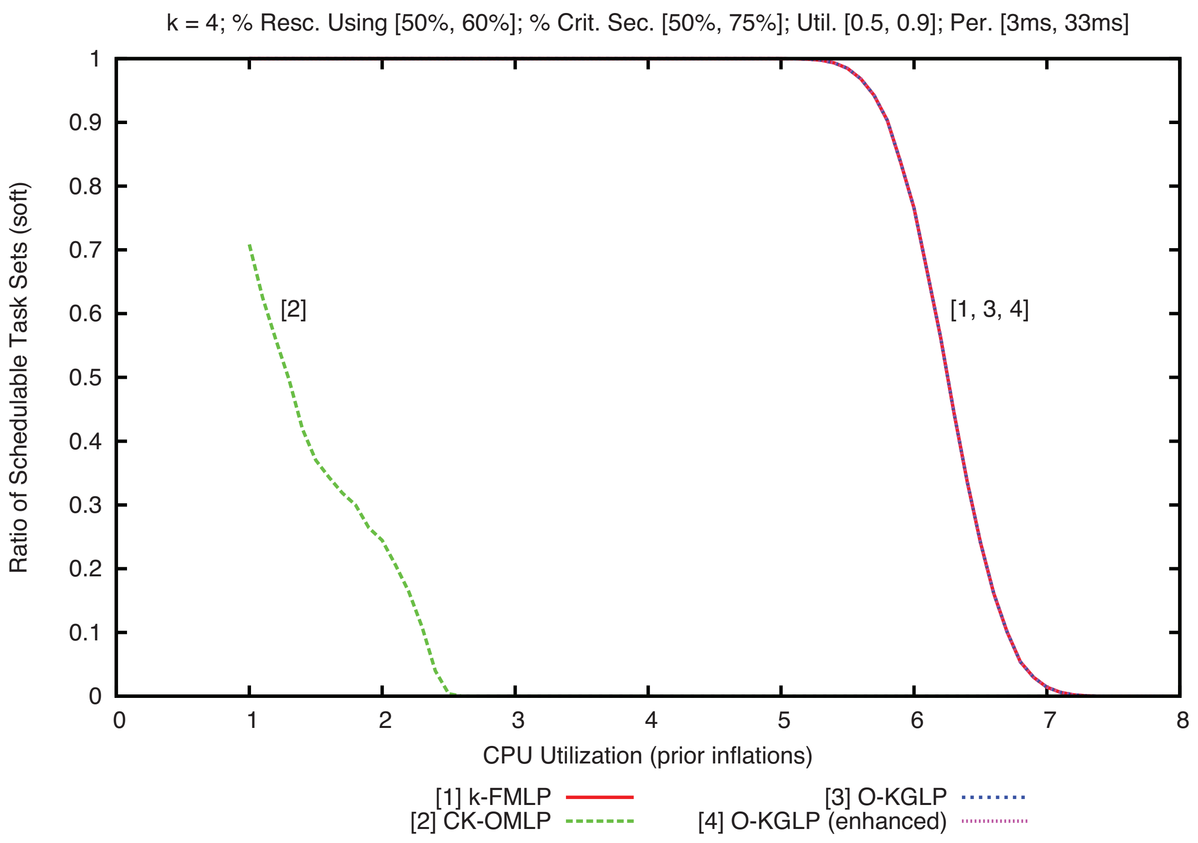

Figure 11: *Schedulability under the* CK-OMLP *is very poor when there are many non-resource using tasks and critical sections are very long. This is the case in this scenario where non-resource-using tasks make up between 40% to 50% of each task set, and critical sections are a large percentage, between 50% and 75%, of execution time of tasks with heavy utilizations.*

Fig. 8. This is because either the k-FMLP or the standard O-KGLP *consistently* outperforms the other in these cases. However, the enhanced O-KGLP offers better schedulability than either of these in Fig. 10, where neither the k-FMLP nor the standard O-KGLP consistently outperforms the other. In this case, some task sets are schedulable under the k-FMLP but not the standard O-KGLP, while other task sets of relatively equal total (non-inflated) utilization are schedulable under the standard O-KGLP but not the k-FMLP. Thus, the schedulability curve of the enhanced O-KGLP does not always tightly track that of either the k-FMLP or standard O-KGLP. This is because the likelihood that a given task set is schedulable under the enhanced O-KGLP is no less than that under the k-FMLP or standard O-KGLP, and may be greater than either.

Observation 3. *The CK-OMLP may outperform both the O-KGLP and k-FMLP when many tasks are resource-using and critical sections are short, but the CK-OMLP performs very poorly, otherwise.* Under the CK-OMLP, resource-using jobs can experience pi-blocking from $2\lceil m/k \rceil - 1$ resource requests and *non*-resource-using jobs can experience pi-blocking from $\lceil m/k \rceil$ resource requests. The CK-OMLP performs poorly when there are a large number of non-resource-using jobs or when critical sections are large since these jobs may also experience pi-blocking. The poor performance of CK-OMLP is evident in Fig. 11.

Recall that our motivation for developing the O-KGLP has been to support multi-GPU systems, where critical sections are large. Thus, the CK-OMLP is not a good choice to protect GPUs in globally-scheduled JLSP systems.

The CK-OMLP can perform well in some scenarios, however. Under the CK-OMLP, pi-blocking of non-resource-using jobs becomes less harmful as their number decreases, and the CK-OMLP outperforms the O-KGLP and k-FMLP when there are a large number of resource-using jobs and critical sections are short, as seen in Fig. 8. The performance of the k-FMLP is poor in this scenario due the the large number of tasks (it is not asymptotically optimal). The O-KGLP performs poorly by comparison since resource-using jobs can be pi-blocked by at most $2\lceil m/k \rceil + 2$ resource requests: three more than under the CK-OMLP.

7 Conclusion

In this paper, we have presented the first real-time k-exclusion locking protocol designed specifically for globally-scheduled JLSP systems. We discussed several approaches to developing an efficient k-exclusion locking protocol and showed that even informed methodology may lead to sub-optimal results. This discussion exposed specific problems that must be resolved by an asymptotically optimal solution. We then developed the O-KGLP and demonstrated that it is asymptotically optimal. Unlike some non-optimal k-exclusion locking protocols, worst-case blocking time under the O-KGLP is not a function of task set size, but rather the number of system CPUs and scales inversely with the number of resources. The O-KGLP improves upon prior k-exclusion locking protocols that are not asymptotically optimal or that adversely affect non-resource-using tasks. We carried out schedulability experiments to compare the O-KGLP and other k-exclusion locking protocols. It was demonstrated that the asymptotic optimality of the O-KGLP often leads to improved schedulability, especially for multi-GPU applications.

In future work, we will evaluate 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. This work will include an implementation of the O-KGLP and a study of its runtime performance. An efficient implementation should use data structures that keep the runtime complexity of enforcing the various rules of the O-KGLP low. Further, this implementation must be done within an operating system environment where there are strict limitations on dynamic memory allocation. This can make the implementation of even relatively simply data structures challenging.

References

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

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

[3] Brandenburg, B.: Scheduling and locking in multiprocessor real-time operating systems. Ph.D. thesis, University of North Carolina at Chapel Hill (2011)

[4] Brandenburg, B., Anderson, J.: Optimality results for multiprocessor real-time locking. In: Proceedings of the 31st IEEE Real-Time Systems Symposium, pp. 49–60 (2010)

[5] Brandenburg, B., Anderson, J.: Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In: Proceedings of the 11th International Conference on Embedded Software, pp. 69–78 (2011)

[6] Chen, M.I.: Schedulability analysis of resource access control protocols in real-time systems. Ph.D. thesis, University of Illinois at Urbana-Champaign (1992)

[7] Devi, U., Anderson, J.: Tardiness bounds under global EDF scheduling on a multiprocessor. In: Real-Time Systems, vol. 38, pp. 133–189 (2008)

[8] Dhall, S.K., Liu, C.L.: On a real-time scheduling problem. Operations Research **26**(1), 127 (1978)

[9] Elliott, G., Anderson, J.: Globally scheduled real-time multiprocessor systems with GPUs. Real-Time Systems **48**, 34–74 (2012)

[10] Elliott, G., Anderson, J.: Robust real-time multiprocessor interrupt handling motivated by GPUs. In: Proceedings of the 24th Euromicro Conference on Real-Time Systems, pp. 267–276 (2012)

[11] Erickson, J., Devi, U., Anderson, J.: Improved tardiness bounds for Global EDF. In: Proceedings of the 22nd EuroMicro Conference on Real-Time Systems, pp. 14–23 (2010)

[12] Fischer, M., Lynch, N., Burns, J., Borodin, A.: Distributed FIFO allocation of identical resources using small shared space. ACM Transactions on Programming Languages and Systems **11**(1), 90–114 (1989)

[13] Gai, P., Abeni, L., Buttazzo, G.: Multiprocessor DSP scheduling in system-on-a-chip architectures. In: Proceedings of the 14th EuroMicro Conference on Real-Time Systems, pp. 231–238 (2002)

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

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

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

[17] Raynal, M., Beeson, D.: Algorithms for mutual exclusion. MIT Press (1986)

A Algorithms for Detailed Blocking Analysis

This appendix presents pseudocode for the algorithms used to compute blocking terms for the k-FMLP, O-KGLP, and CK-OMLP in Secs. 5 and 6. The algorithms make use of tardiness bounds to provide support for soft real-time blocking analysis. It is assumed that variables T, T^R, m, and k are in a global scope and are thus not passed as routine parameters. It is also assumed that attributes of task T_i, such as p_i, e_i, l_i, and b_i, are in scope with T_i. That is, for example, b_i is short-hand for $T_i.b$ or $b(T_i)$.

Algorithm 1 computes blocking terms for the O-KGLP and is derived from the detailed analysis presented in Sec. 5. Algorithm 2 computes blocking terms for the k-FMLP and is derived from the online appendix of [10]. Finally, Algorithm 3 computes blocking terms for the CK-OMLP and is derived from [5].

Algorithm 1 Calculates b_i under the O-KGLP for each task in task set.

```
1: procedure OKGLP-COMPUTE-BLOCKING
2:     for each $T_i \in T$ do
3:         if $T_i \notin T^R$ then
4:             $b_i \leftarrow 0$                                        ▷ $T_i$ is not resource-using.
5:         else if $\left|T^R\right| \leq k$ then
6:             $b_i \leftarrow 0$                                        ▷ Request trivially satisfied.
7:         else
8:             $criticalSections \leftarrow [\,]$
9:             $numSections \leftarrow 0$
10:            if $\left|T^R\right| \leq m + k$ then
11:                for each $T_j \in T^R \setminus \{T_i\}$ do
12:                    $criticalSections[numSections] \leftarrow l_j$
13:                    $numSections \leftarrow numSections + 1$
14:                $maxTerms \leftarrow \left\lfloor \frac{\left|T^R\right|-1}{k} \right\rfloor$                  ▷ Equation (7).
15:            else
16:                for each $T_j \in T^R \setminus \{T_i\}$ do
17:                    $tif \leftarrow \left\lceil \frac{p_i+x_i+p_j+x_j}{p_j} \right\rceil$                ▷ Equation (9).
18:                    for $c \leftarrow 0$ to $tif - 1$ do
19:                        $criticalSections[numSections] \leftarrow l_j$
20:                        $numSections \leftarrow numSections + 1$
21:                $maxTerms \leftarrow 2\left\lceil \frac{m}{k} \right\rceil + 2$                   ▷ Equation (6).
22:            SORT-DESCENDING($criticalSections$)
23:            $terms \leftarrow$ MIN($maxTerms, numSections$)
24:            $b_i \leftarrow 0$
25:            for $c \leftarrow 0$ to $terms - 1$ do
26:                $b_i \leftarrow b_i + criticalSections[c]$
```

Algorithm 2 Calculates b_i under the k-FMLP for each task in task set.

```
1: procedure KFMLP-Compute-Blocking
2:     for each $T_i \in T$ do
3:         if $T_i \notin T^R$ then
4:             $b_i \leftarrow 0$                                    ▷ $T_i$ is not resource-using.
5:         else if $\left|T^R\right| \leq k$ then
6:             $b_i \leftarrow 0$                                    ▷ Request trivially satisfied.
7:         else
8:             $\mathit{criticalSections} \leftarrow [\,]$
9:             $\mathit{numSections} \leftarrow 0$
10:            for each $T_j \in T^R \backslash \{T_i\}$ do
11:                $\mathit{criticalSections}[\mathit{numSections}] \leftarrow l_j$
12:                $\mathit{numSections} \leftarrow \mathit{numSections} + 1$
13:            Sort-Descending($\mathit{criticalSections}$)
14:            $\mathit{terms} \leftarrow \left\lfloor \frac{|T^R|-1}{k} \right\rfloor$
15:            $b_i \leftarrow 0$
16:            for $c \leftarrow 0$ to $\mathit{terms} - 1$ do
17:                $b_i \leftarrow b_i + \mathit{criticalSections}[c]$
```

Algorithm 3 Calculates b_i under the CK-OMLP for each task in task set.

```
    ▷ Blocking computed in two phases: resource blocking and then donation blocking.
    ▷ $br_i$ and $bd_i$ denotes resource and donation blocking for task $T_i$, respectively.
 1: procedure CKOMLP-Compute-Blocking
 2:     for each $T_i \in T$ do
 3:         CKOMLP-Resource-Blocking($T_i$)
 4:         CKOMLP-Donation-Blocking($T_i$)
 5:         $b_i \leftarrow br_i + bd_i$

    ▷ Resource blocking only affects resource-using tasks.
 6: procedure CKOMLP-Resource-Blocking($T_i$)
 7:     if $T_i \notin T^R$ then
 8:         $br_i \leftarrow 0$                                        ▷ $T_i$ is not resource-using.
 9:     else if $\left|T^R\right| \leq k$ then
10:         $br_i \leftarrow 0$                                        ▷ Request trivially satisfied.
11:     else
12:         $criticalSections \leftarrow [\,]$
13:         $numSections \leftarrow 0$
14:         for each $T_j \in T^R \backslash \{T_i\}$ do
15:             $tif \leftarrow$ Min($\left\lceil \frac{p_i+x_i+p_j+x_j}{p_j} \right\rceil$, 2)     ▷ Soft real-time, interference capped at 2.
16:             for $c \leftarrow 0$ to $tif - 1$ do
17:                 $criticalSections[numSections] \leftarrow l_j$
18:                 $numSections \leftarrow numSections + 1$
19:         Sort-Descending($criticalSections$)
20:         $terms \leftarrow$ Min($\left\lceil \frac{m}{k} \right\rceil - 1, numSections$)
21:         $br_i \leftarrow 0$
22:         for $c \leftarrow 0$ to $terms - 1$ do
23:             $br_i \leftarrow br_i + criticalSections[c]$

    ▷ Donation blocking affects all tasks.
24: procedure CKOMLP-Donation-Blocking($T_i$)
25:     $doneeDurations \leftarrow [\,]$
26:     $numDurations \leftarrow 0$
27:     for each $T_j \in T^R \backslash \{T_i\}$ do
28:         $doneeDurations[numDurations] \leftarrow br_j + l_j$
29:         $numDurations \leftarrow numDurations + 1$
30:     if $numDurations = 0$ then
31:         $bd_i \leftarrow 0$
32:     else
33:         $bd_i \leftarrow$ Max($doneeDurations$)
```