Concurrency Groups: A New Way to Look at Real-Time Multiprocessor Lock Nesting

Catherine E. Nemitz, Tanya Amert, Manish Goyal, and James H. Anderson
The University of North Carolina at Chapel Hill
{nemitz,tamert,manishg,anderson}@cs.unc.edu

ABSTRACT

When designing a real-time multiprocessor locking protocol, the allowance of lock nesting creates complications that can kill parallelism. Such protocols are typically designed by focusing on the arbitration of resource requests that should be *prohibited* from executing concurrently. This paper proposes "concurrency groups," a new concept that reflects an alternative point of view that focuses instead on requests that can be *allowed* to execute concurrently. A *concurrency group* is simply a group of lock requests, determined offline, that can safely execute together. This paper's main contribution is the CGLP, a new real-time multiprocessor locking protocol that supports lock nesting through the use of concurrency groups. The CGLP is able to reap runtime parallelism benefits that have eluded prior protocols by investing effort offline in the construction of concurrency groups. A schedulability study is presented to quantify such benefits, as well as an efficient approach to determining such groups using an Integer Linear Program (ILP) solver.

CCS CONCEPTS

• **Computer systems organization → Real-time systems**; *Embedded and cyber-physical systems*; • **Software and its engineering → Mutual exclusion**; **Real-time systems software**; **Synchronization**; *Process synchronization.*

KEYWORDS

multiprocess locking protocols, nested locks, priority-inversion blocking, real-time locking protocols

ACM Reference Format:
Catherine E. Nemitz, Tanya Amert, Manish Goyal, and James H. Anderson. 2019. Concurrency Groups: A New Way to Look at Real-Time Multiprocessor Lock Nesting. In *27th International Conference on Real-Time Networks and Systems, November 06–08, 2019, Toulouse, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Work supported by NSF grants CNS 1409175, CNS 1563845, CNS 1717589, and CPS 1837337, ARO grant W911NF-17-1-0294, and funding from General Motors. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGS-1650116. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

RTNS '19, November 06–08, 2019, Toulouse, France

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

While real-time multiprocessor locking protocols have been studied for over thirty years [20], the issue of enabling unrestricted *lock nesting—i.e.,* a task holding locks on several resources simultaneously—in an efficient manner was considered only relatively recently [23]. The desire to support nesting is motivated by practical concerns: use cases are common in practice in which a task must access multiple resources at once without interference from other tasks. However, unrestricted lock nesting causes complications in real-time systems.

Many of these complications are rooted in the fact that it is difficult to avoid negating the parallelism that the underlying hardware platform affords. This difficulty is due, at least in part, to two fundamental problems. The first is a problem we call the *Transitive Blocking Chain Problem*: when lock nesting is allowed, chains of requests can form that prevent resource requests from being satisfied even though the requested resources are free. The second is a problem we call the *Request Timing Problem*: even in protocols designed to reap gains in parallelism, such gains can be negated by even small variations in resource request durations or other timing details. All existing real-time multiprocessor locking protocols that allow nesting are subject to one or both of these problems.

In this paper, we present the CGLP, the first ever protocol designed to address both problems. The design of the CGLP reflects a fundamentally different approach compared to prior work: *rather than viewing a locking protocol as merely* ***preventing*** *resources from being accessed concurrently, we instead view it as a mechanism that* ***safely allows*** *concurrency with respect to shared resources.* Doing so allows us to take advantage of the timing information provided in real-time systems to gain parallelism; this is reflected in the determination of per-request blocking bounds (which are used in schedulability analysis). The CGLP is designed around a new notion: groups of tasks called *concurrency groups* that may safely execute concurrently.

Before describing the CGLP further, we first describe the two fundamental problems noted above in more detail.

Transitive blocking chain problem. Most approaches to coordinating resource accesses order requests using a pre-determined scheme such as first-in-first-out (FIFO), which we assume here. Any such scheme can result in chains of requests all blocked on a single request. Such a *transitive blocking chain* can cause a request to be blocked by another request with no resources in common. This problem can affect both nested and non-nested requests. We illustrate it via an example involving only nested requests.

Example 1.1. Consider a scenario with six tasks and seven resources, ℓ_a through ℓ_g. Each task τ_i issues a single request, $\mathcal{R}_i$, for two resources for some duration. In Fig. 1, resources are shown along the horizontal axis, and requests have enqueued in task-index

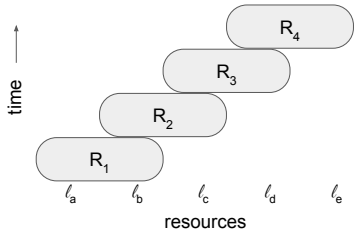


Figure 1: FIFO-ordering.

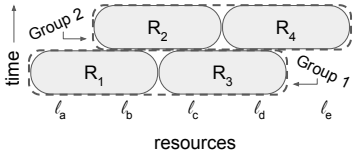


Figure 2: Optimized offline ordering.

order. The maximum duration of each request is illustrated by a box of that height numbered with the corresponding task's index. In Fig. 1, $\mathcal{R}_1$ holds ℓ_a and ℓ_b. This prevents $\mathcal{R}_2$ from acquiring ℓ_b and ℓ_c. Thus, $\mathcal{R}_2$ is blocked by $\mathcal{R}_1$. A transitive blocking chain may form, as shown in Fig. 1. Such a chain causes $\mathcal{R}_4$ to experience blocking for up to the duration of three critical-section lengths.

When determining schedulability, we must account for the worst-case ordering of request execution to calculate the worst-case blocking of each task. The ordering in Fig. 1 illustrates the chain that causes the worst-case blocking for $\mathcal{R}_4$.

Example 1.1 (continued). To solve the Transitive Blocking Chain Problem, the CGLP partitions the requests in Fig. 1 into two groups wherein concurrent execution is allowed, as shown in Fig. 2. At runtime, resource access is provided on a per-group basis. As seen in Fig. 2, doing so prevents transitive blocking chains from forming.

We call groups of tasks as just described *concurrency groups*. Such groups are determined offline based on task-system characteristics.

Request timing problem. Although existing approaches have addressed the Transitive Blocking Chain Problem [13, 17], worst-case blocking under these approaches is heavily dependent on the timing of request issuances and differences in request durations. Such timing-related variations can cause "gaps" in the underlying queues utilized by a protocol. Such gaps inhibit parallel execution.

Example 1.2. Consider requests $\mathcal{R}_1$–$\mathcal{R}_4$, shown in Fig. 3, issued in numerical order and enqueued. $\mathcal{R}_5$ is then issued. By any protocol known to us, $\mathcal{R}_5$ will be enqueued after $\mathcal{R}_4$. Another "slot" that could have been considered is shown in Fig. 3, but $\mathcal{R}_5$ cannot be inserted here, as this would further delay $\mathcal{R}_4$. (Such delays are problematic because the number of later-arriving requests is generally unbounded.) Observe how the timing of the issuance of $\mathcal{R}_2$ caused a gap just after time 30 into which no conflicting request can fit.

The CGLP obviates such gaps by using task-system characteristics to pre-determine the "slots" into which requests are inserted. Because this determination is made offline, it is not subject to runtime timing variations.

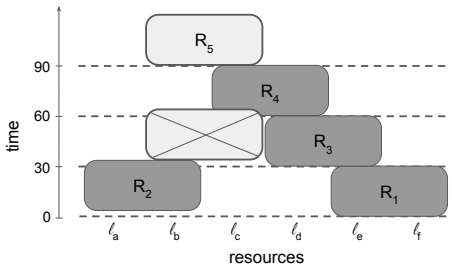


Figure 3: An illustration of the Request Timing Problem. $\mathcal{R}_5$ may not be inserted in the earlier slot marked by an 'X', as this would delay an already issued request.

In many protocols, having to deal with requests of different durations can also cause "gaps" similar to that in Ex. 1.2. Thus, such differences are also a source of the Request Timing Problem. The concurrency groups of the CGLP are constructed so as to minimize such differences and thus eliminate these gaps.

Contributions. We introduce a new real-time multiprocessor locking protocol, the CGLP, that allows lock nesting and that results in lower blocking and overhead than prior protocols for many systems. We gain analytical advantages by focusing on which tasks may execute requests concurrently.

The CGLP has an offline component for determining concurrency groups that simplifies the arbitration of requests at runtime. This component examines various optimizations to the request ordering that would be impractical to explore at runtime. We begin by framing the construction of concurrency groups as a graph-coloring problem and then explore approaches for determining groups that improve worst-case blocking bounds. To assess the CGLP, we conducted a schedulability study, the results of which are presented herein. We also present a fast approach for determining concurrency groups using an Integer Linear Program (ILP) solver.

Organization. We begin with necessary background in Sec. 2. In Sec. 3, we introduce the CGLP by first presenting a basic variant of it and an analysis of its blocking complexity. We then consider various extensions to the protocol in Sec. 4. We present the aforementioned schedulability study in Sec. 5 and conclude in Sec. 6.

2 BACKGROUND

Before summarizing prior work on real-time locking protocols for multiprocessor systems, we provide necessary details of our task and resource models.

System model. We focus on a sporadic task set Γ comprised of n tasks $\{\tau_1..\tau_n\}$ on a multiprocessor platform with m processors. We assume these tasks are scheduled with a job-level fixed-priority scheduler such as Global Earliest Deadline First (G-EDF).

Resource model. When a task requires access to one or more resources, it *issues* a request. We denote an arbitrary request as $\mathcal{R}_i$ and an arbitrary resource as ℓ_a. We say a request $\mathcal{R}_i$ is *satisfied* when it *holds* all of its required resources, denoted D_i.[1] $\mathcal{R}_i$ executes its critical section for at most L_i time units before it *completes* and

[1]We assume the use of dynamic group locks [23], which coalesce all resources a task may require concurrently under a single request. For example, if a task requires access

releases all of its held resources. A request is *active* from the time it is issued to the time it completes. The maximum critical-section length of any request is denoted L_{max}. We call a request $\mathcal{R}_i$ a *write* request if it requires mutually exclusive access to D_i or a *read* request if other requests may access D_i concurrently.

A particular challenge is allowing nested resource access, in which a task holds multiple resources concurrently. We focus primarily on providing efficient synchronization for nested write requests; other work has presented methods for efficiently handling read requests and non-nested requests in the presence of write requests and nested requests [17]. We also consider how our protocol can be extended to accommodate read requests.

We measure efficiency with regard to reducing the delays lower-priority tasks cause for higher-priority tasks. Specifically, we look at *priority-inversion blocking* (pi-blocking), the delay a task incurs due to waiting for access to one or more resources held by a lower-priority task. Achieving a reduction in pi-blocking ought to be done with minimal introduction of additional overhead. In this paper, we focus on locking protocols that are spin-based; a task busy-waits until its request is satisfied.

Prior work. There is a large body of work aimed at locking protocols for multiprocessor systems. However, few of these approaches allow for nesting. We focus on those that do. One synchronization approach that allows nested access to resources is the *multiprocessor bandwidth inheritance protocol* (M-BWI) [8, 9]. Another approach is *MrsP* [6, 10, 27]. Rather than using dynamic group locks, both the M-BWI and MrsP require an ordering on nested resource acquisition to prevent deadlock.[2] A straightforward bound on the blocking a request may experience when deadlock is prevented by resource ordering is exponential in the number of resources [21]. Computing a tight bound on worst-case blocking is NP-hard when nesting is allowed [26].

The *real-time nested locking protocol (RNLP)* [13, 16, 17, 22–25] family of protocols also supports nested requests, and each protocol uses dynamic group locks. Of the protocols mentioned, most do not handle the Transitive Blocking Chain Problem. Those that do are the fast RW-RNLP [17] and the C-RNLP [13]. The fast RW-RNLP eliminates transitive blocking chains for non-nested requests (and read requests) by ensuring that they are enqueued in separate data structures from nested requests. (Non-nested requests can also experience increased blocking due to transitive blocking chains. For example, a request for $\{\ell_e\}$ issued after $\mathcal{R}_4$ in Ex. 1.1 would be blocked for up to $4 \cdot L_{max}$ time units.) Only once requests are at the head of their respective queue(s) do they compete for resources; it is not possible for a chain of blocking to impact a non-nested (or read) request. Nested write requests, however, may still suffer from transitive blocking chains under the fast RW-RNLP.

To our knowledge, the C-RNLP is the only protocol that breaks transitive blocking chains for nested write requests. To do so, when any request $\mathcal{R}_i$ is issued, all other active requests must be evaluated to determine the earliest spot in the queues corresponding to D_i in which $\mathcal{R}_i$ may cut ahead without increasing blocking times for other requests. This requires the maintenance of a significant amount of state, which can be detrimental to the protocol's performance. Existing implementations require a mutex to ensure safe, atomic insertion into all the maintained queues that are required.

We present the CGLP, which builds on the notion of a reader-reader locking protocol (a synchronization mechanism that manages resource access between groups of read requests [16]). The CGLP can be implemented without the use of a mutex and allows one group of requests access at a time; any requests from another group must wait until the satisfied requests complete. In this sense, the protocol alternates between phases in which different groups of requests are satisfied. This reader-reader paradigm is an extension of the R^{3}LP [17], which coordinates three groups of read requests. Existing work [16, 17] has also explored layering synchronization mechanisms to first establish that some group of requests does not overlap, and thus could be viewed as a group of read requests relative to each other.

The CGLP is motivated by the current lack of a solution to the Request Timing Problem. Existing protocols miss opportunities for concurrent execution because of these timing issues. This occurs based on the design of these protocols, which is based on the notion of which requests must be *prevented* from executing concurrently. Our new approach groups requests that are *allowed* to execute concurrently. These groups are established by using a graph coloring approach. Such an approach has been used to solve a variety of other resource allocation problems [2, 3, 7, 14].

3 CONCURRENCY GROUPS

We develop the Concurrency Group Locking Protocol (CGLP) to address both the Transitive Blocking Chain Problem and the Request Timing Problem. Recall the pathological case of transitive blocking presented in Sec. 1. Although each nested request required only two resources, a FIFO-ordered synchronization protocol could cause a long chain of transitive blocking, as illustrated in Fig. 1. The blocking chain in this example could be eliminated by partitioning the requests into the two groups shown in Fig. 2 and allowing only one group to execute at any given time. This captures the basic intuition of the CGLP; the protocol is described in detail below.

In this section, we begin by discussing how to generate concurrency groups for an arbitrary set of write requests. Then we show how phase-based access to resources can be achieved by generalizing a phase-based protocol. We finish this section by showing how the CGLP can address the Request Timing Problem.

3.1 Offline Group Creation via Graph Coloring

The Vertex Coloring Problem entails finding the minimum number of colors, k, with which the vertices of a graph can be colored such that no adjacent vertices have the same color. A graph that requires at most k colors is said to be *k-colorable*. Given a set of write requests, we seek to create concurrency groups. All requests in a single group must not share any resources. Our goal is to create the minimum number of groups, as this maximizes the possible concurrency. We transform our problem to the Vertex Coloring Problem in two steps. First, for each request $\mathcal{R}_i$, we create a corresponding vertex $\mathcal{S}_i$. Once we have added all vertices to the graph, we add edges. An edge is added between $\mathcal{S}_i$ and $\mathcal{S}_j$, where $i \neq j$, if $D_i \cap D_j \neq \emptyset$.

to ℓ_a and then conditionally requires access to either ℓ_b or ℓ_c, it issues a single request for $\{\ell_a, \ell_b, \ell_c\}$.

[2] This ordering refers to the order in which resources must be acquired by a given task, not the order in which requests are satisfied [12].

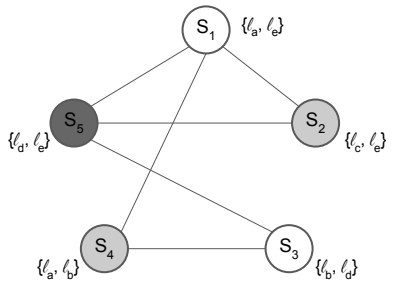


Figure 4: An example coloring.

Example 3.3. Consider a task set that produces five requests: $\mathcal{R}_1$ for $D_1 = \{\ell_a, \ell_e\}$, $\mathcal{R}_2$ for $D_2 = \{\ell_c, \ell_e\}$, $\mathcal{R}_3$ for $D_3 = \{\ell_b, \ell_d\}$, $\mathcal{R}_4$ for $D_4 = \{\ell_a, \ell_b\}$, and $\mathcal{R}_5$ for $D_5 = \{\ell_d, \ell_e\}$. The graph representation of these requests is shown in Fig. 4. For example, $\mathcal{S}_4$ is connected to $\mathcal{S}_1$ and $\mathcal{S}_3$ because $D_4 \cap D_1 = \{\ell_a\}$ and $D_4 \cap D_3 = \{\ell_b\}$. $\mathcal{S}_4$ does not have an edge to either $\mathcal{S}_2$ or $\mathcal{S}_5$, as $D_4 \cap D_2 = \emptyset$ and $D_4 \cap D_5 = \emptyset$.

To determine the minimum number of concurrency groups, we find the minimum k such that the graph can be colored with k colors. This results in k groups, $\mathcal{G}_1$ through $\mathcal{G}_k$. A specific coloring informs which requests belong in which group; if a vertex $\mathcal{S}_i$ is assigned color g, $\mathcal{R}_i \in \mathcal{G}g$.

Example 3.3 (continued). This graph is 3-colorable, so only three concurrency groups are required. In particular, we can color the vertices as shown in Fig. 4, which results in $\mathcal{G}_1 = \{\mathcal{R}_1, \mathcal{R}_3\}$, $\mathcal{G}_2 = \{\mathcal{R}_2, \mathcal{R}_4\}$, and $\mathcal{G}_3 = \{\mathcal{R}_5\}$.

By our construction of the graph and the constraints on a solution to the Vertex Coloring Problem, none of the requests in a given concurrency group require any overlapping resources. As is standard for the analysis of real-time systems, we assume that all possible requests are known *a priori*. Thus, we can run a k-colorability analysis offline to determine the number of groups required for a given system and add each request to a group based on its assigned color. Though the Vertex Coloring Problem is NP-hard, we shown in Sec. 5 that, for many systems, groups can be determined in a reasonable amount of time. What remains is to coordinate access to these groups of requests during runtime.

3.2 Group Arbitration

Arbitration among concurrency groups must occur online. At most one group may be allowed to be satisfied at a time. All requests in a given group may run concurrently with each other, but requests from different groups must not be allowed to execute together.

In this way, requests within the same group may be considered to be read requests relative to each other. Thus, we must provide synchronization between k groups of readers. We do so with a protocol called the R^{k}LP, which we present as a k-phased extension to the 2-phased [16] and 3-phased [17] reader-reader locking protocols.

Example 3.3 (continued). $\mathcal{R}_1$ and $\mathcal{R}_3$, both in $\mathcal{G}_1$, do not share resources, so no synchronization protection is required between them. However, $\mathcal{G}_1$ and $\mathcal{G}_2$ cannot be allowed to execute concurrently.

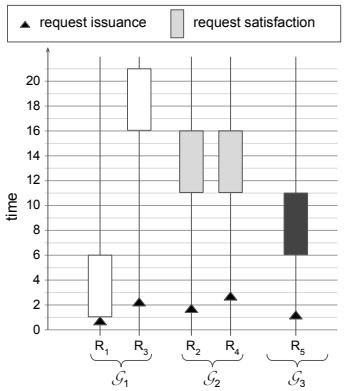


Figure 5: Trace of executions of requests.

To refine how we reason about the R^{k}LP, we present a series of rules that encapsulate how this protocol functions. We call the time during which a group is active a *phase*.

- **G1** Each group is either *active*, *waiting*, or *inactive*, and at most one group is active at any time.
- **G2** If a request belonging to an inactive group is issued, then the group becomes active if no group is active, or waiting if there is an active group.
- **G3** A waiting group becomes active once all groups that were active or waiting when this group entered the waiting state have completed a single phase of execution.
- **G4** All active requests in a group that becomes active are satisfied immediately.

Example 3.3 (continued). As depicted in Fig. 5, $\mathcal{R}_1$ is issued at time $t = 1$. Because no other groups are active at $t = 1$, $\mathcal{G}_1$ becomes active immediately, by Rule G2. By Rule G4, $\mathcal{R}_1$ is satisfied immediately. At $t = 1.5$, $\mathcal{R}_5$ is issued. At most one group can be active at any time and $\mathcal{G}_1$ is still active, so $\mathcal{G}_3$ is now waiting, by Rules G1 and G2. By Rules G3 and G4, $\mathcal{R}_5$ will be satisfied when $\mathcal{G}_1$ has completed a phase of execution. This occurs at time $t = 6$.

- **G5** All requests satisfied in a phase finish by the end of that phase.
- **G6** When all satisfied requests of a phase finish, the group enters the waiting state if there are any active requests in the group. Otherwise it enters the inactive state.
- **G7** When all satisfied requests of a phase finish, the completion of the last request and the transition to a new active phase, if there was a waiting group, happen atomically.

Example 3.3 (continued). $\mathcal{G}_3$ is active from $t = 6$ to $t = 11$. $\mathcal{R}_5$ completes by the end of that phase, by Rule G5. When $\mathcal{R}_5$ completes, $\mathcal{G}_3$ becomes inactive, by Rule G6. At that time, $\mathcal{G}_2$ becomes active, by Rules G3 and G7.

- **G8** If a request belonging to the active group is issued while the group is active, it becomes satisfied immediately as part of the current phase only if there are no waiting groups. (If there is a waiting group, it will be satisfied in the next active phase of its group.)

Example 3.3 (continued). $\mathcal{R}_3$ is issued at time $t = 2.5$, while $\mathcal{G}_1$ is active and there are waiting groups, so $\mathcal{R}_3$ must wait for the next active phase of $\mathcal{G}_1$, by Rule G8.

The above rules capture how the k concurrency groups alternate between active phases. We discuss our spin-based implementation of the R^{k}LP in an online appendix [18].

3.3 Bounding Blocking

The essential component to determining schedulability given a locking protocol is the bound on worst-case pi-blocking. With the R^{k}LP, the bound depends on the time it takes each of the k groups to execute. Intuitively, each phase may execute for up to the maximum critical-section length, L_{max}. Below, we show a bound on the worst-case acquisition delay.

Lemma 3.1. *When there is at least one waiting group, the current phase of the active group ends within L_{max} time units.*

Proof. When there is at least one waiting group, newly issued requests belonging to the active group are not immediately satisfied, by Rule G8. Therefore, only the currently satisfied requests must complete before the active group enters the waiting state. Any satisfied request executes for at most L_{max} time units. Thus, the current phase of the active group will end within L_{max} time units, and the active group will become waiting or inactive. □

Theorem 3.1. *In a system with k concurrency groups, a request $\mathcal{R}_i$ has a maximum acquisition delay of $k \cdot L_{max}$.*

Proof. Upon being issued, if request $\mathcal{R}_i$ belonging to $\mathcal{G}_g$ is not satisfied immediately, then at least one group is waiting, by Rules G2 and G8. Furthermore, $\mathcal{G}_g$ is either waiting or active.

Suppose $\mathcal{G}_g$ is waiting. Some other group must be active, by Rule G2. Because there is a waiting group ($\mathcal{G}_g$), the active group will complete within L_{max} time units, by Lemma 3.1. By Rule G3, $\mathcal{G}_g$ will become active once all groups that were active or waiting when $\mathcal{G}_g$ entered the waiting state have completed a single phase of execution. Because there are at most k concurrency groups, at most $k-1$ other groups could have been active or waiting when $\mathcal{G}_g$ entered the waiting state. Thus, at most $k-1$ other groups must complete a phase, and each phase will last for at most L_{max} time units. Hence, the maximum acquisition delay for $\mathcal{R}_i$ is $k-1 \cdot L_{max}$ in this case. (By Rule G4, as soon as $\mathcal{G}_g$ becomes active, $\mathcal{R}_i$ will be satisfied.)

Suppose instead that $\mathcal{G}_g$ is active. Because $\mathcal{R}_i$ is not satisfied immediately, there must be a waiting group (preventing $\mathcal{R}_i$ from being satisfied immediately due to Rule G8). $\mathcal{G}_g$ will complete its active phase within L_{max} time units. Its group will then transition to the waiting state by Rule G6. As reasoned above, the waiting $\mathcal{G}_g$ will become active, and thus $\mathcal{R}_i$ be satisfied, within $k-1 \cdot L_{max}$ time units. Thus, in total, the worst-case acquisition delay for $\mathcal{R}_i$ is $k \cdot L_{max}$ time units. □

We revisit our example to see that this blocking bound is tight.

Example 3.3 (continued). When $\mathcal{R}_3$ is issued at $t = 2.5$ in Fig. 5, it cannot be satisfied immediately, by Rule G8. Its maximum acquisition delay is $3 \cdot L_{max}$, corresponding to a phase of each of $\mathcal{G}_1$, $\mathcal{G}_3$, and $\mathcal{G}_2$, as illustrated in Fig. 5.

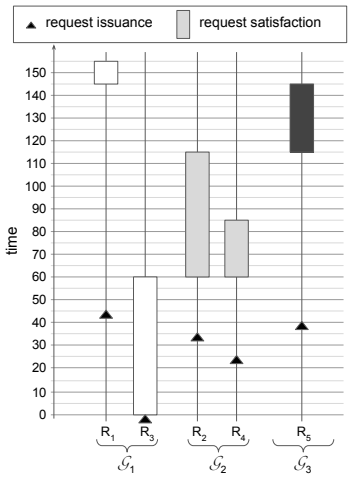


Figure 6: An illustration of the maximum blocking for $\mathcal{R}_1$.

3.4 Refining the Blocking Bound

Up to this point, we have not specified the critical-section lengths, so we treated each as L_{max}. In this section, we focus on the benefits of allowing critical-section lengths to factor into the group assignments. When requests have varying critical-section lengths, the bound in Theorem 3.1 may be overly pessimistic. When analyzing the impact of each concurrency group on the blocking a given request may experience, we define the maximum critical-section length of a group $\mathcal{G}_g$ to be $L_{max}^{\mathcal{G}_g}$.

Example 3.3 (continued). Let the critical-section lengths of the five requests be $L_1 = 10$, $L_2 = 55$, $L_3 = 60$, $L_4 = 25$, and $L_5 = 30$ time units. Then, $L_{max}^{\mathcal{G}_1} = 60$, $L_{max}^{\mathcal{G}_2} = 55$, and $L_{max}^{\mathcal{G}_3} = 30$.

Lemma 3.2. *When there is at least one waiting group, the current phase of the active group $\mathcal{G}_g$ ends within $L_{max}^{\mathcal{G}_g}$ time units.*

Proof. As in Lemma 3.1, when at least one group is waiting, no new requests belonging to $\mathcal{G}_g$ may be satisfied. Thus, the current phase of $\mathcal{G}_g$ will end once all satisfied requests complete, the maximum duration of which is $L_{max}^{\mathcal{G}_g}$. □

Theorem 3.2. *The acquisition delay a request $\mathcal{R}_i$ may experience is at most $\sum_{c=1}^{k} L_{max}^{\mathcal{G}_c}$ time units.*

Proof. As in Theorem 3.1, $\mathcal{R}_i$ may need to wait for the completion of at most one phase of each of the k groups, including its own, before being satisfied. Thus, the maximum acquisition delay of $\mathcal{R}_i$ is $\sum_{c=1}^{k} L_{max}^{\mathcal{G}_c}$. □

Example 3.3 (continued). Consider the execution trace shown in Fig. 6. In this trace, $\mathcal{R}_1$ is released at $t = 45$ and satisfied at time $t = 145$, so it is blocked for 100 time units. By Theorem 3.2, the worst-case blocking of $\mathcal{R}_1$ is 60 + 55 + 30 = 145 time units. Note that this is far less time than the $3 \cdot 60 = 180$ time units given as a bound by Theorem 3.1.

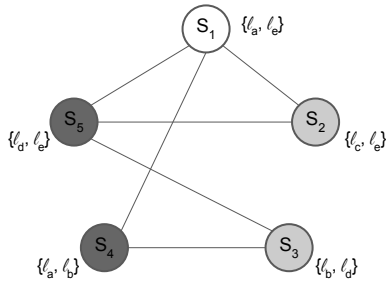


Figure 7: An alternate coloring.

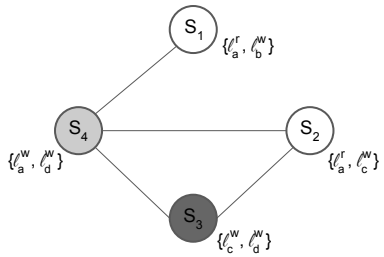


Figure 8: Graph of mixed-type requests.

4 EXTENSIONS

Now that we have explained the fundamental components to the CGLP, we discuss several extensions to the protocol, such as performance improvements and the handling of both read and write requests. For a given set of requests, it can also be beneficial to explore alternate concurrency groupings or optimizations based on the number of requests per resource.

4.1 Alternate Coloring Choices

In the basic version of the CGLP, we picked an arbitrary coloring of the vertices that required the minimum number of colors. However, there can be multiple ways to color a set of vertices with k colors, resulting in different concurrency groups.

Example 4.3 (continued). Continuing the running example from the prior section, there are multiple ways of forming three concurrency groups for this set of requests. For example, instead of the coloring shown in Fig. 4, the coloring shown in Fig. 7 would yield $\mathcal{G}_1 = \{\mathcal{R}_1\}$, $\mathcal{G}_2 = \{\mathcal{R}_2, \mathcal{R}_3\}$, and $\mathcal{G}_3 = \{\mathcal{R}_4, \mathcal{R}_5\}$.

As an extension to the basic CGLP, the concurrency groups should be chosen in a manner that minimizes blocking. This can be done by considering the critical-section lengths in light of the blocking bound given in Theorem 3.2 when assigning groups.

Example 4.3 (continued). By Theorem 3.2, the worst-case blocking of any of the requests under the grouping shown in Fig. 4 is $60 + 55 + 30 = 145$ time units. In contrast, the blocking under the grouping of Fig. 7 is at most $10 + 60 + 30 = 100$ time units. Therefore, the grouping shown in Fig. 7 should be used instead of that in Fig. 4.

Ex. 3.3 highlights the improvements in worst-case blocking that can be achieved by creating concurrency groups based on the critical-section lengths of the requests.

4.2 Mixed-Type Requests

A mixed-type request is one in which the task requires write access for one or more resources and only requires read access for some resources. Such a request may occur when a task must read one or more values from various buffers or sensors before writing a resulting computation to some other region of shared memory. We capture these different synchronization requirements in a manner that allows us to exploit the relaxed resource sharing assumptions for read requests. We do so by modifying how we generate the graph corresponding to the requests.

A vertex is created for each request, as before. However, the addition of edges is changed to reflect this different sharing paradigm. When listing the set of resources D_i required by a request $\mathcal{R}_i$, we denote the type of access required (read or write) with a superscript. For example, $D_i = \{\ell_a^r, \ell_b^w\}$ indicates that $\mathcal{R}_i$ requires read access to ℓ_a and write access to ℓ_b.

Example 4.4. Consider a set of requests $\mathcal{R}_1$ through $\mathcal{R}_4$, which require resources $D_1 = \{\ell_a^r, \ell_b^w\}$, $D_2 = \{\ell_a^r, \ell_c^w\}$, $D_3 = \{\ell_c^w, \ell_d^w\}$, and $D_4 = \{\ell_a^w, \ell_d^w\}$. Here, $\mathcal{R}_1$ and $\mathcal{R}_2$ are mixed-type requests and $\mathcal{R}_3$ and $\mathcal{R}_4$ are write requests.

We define $D_i^w = \{\ell_y | \ell_y^w \in D_i\}$ as the set of resources to which $\mathcal{R}_i$ requires write access. An edge is added between two vertices corresponding to requests $\mathcal{R}_i$ and $\mathcal{R}_j$ if $\mathcal{R}_i \neq \mathcal{R}_j$ and $D_i^w \cap D_j \neq \emptyset \vee D_i \cap D_j^w \neq \emptyset$.

Example 4.4 (continued). The graph corresponding to this set of requests is shown in Fig. 8. Here, $D_1^w = \{\ell_b\}$. Although both $\mathcal{R}_1$ and $\mathcal{R}_2$ require ℓ_a, both read ℓ_a: when comparing $\mathcal{R}_1$ and $\mathcal{R}_2$, we check $\{\ell_b\} \cap \{\ell_a, \ell_c\} = \emptyset$ and $\{\ell_c\} \cap \{\ell_a, \ell_b\} = \emptyset$, so no edge is added between $\mathcal{S}_1$ and $\mathcal{S}_2$. This fits the intuition that $\mathcal{R}_1$ and $\mathcal{R}_2$ could be satisfied concurrently. For $\mathcal{R}_1$ and $\mathcal{R}_4$, $\{\ell_a, \ell_b\} \cap \{\ell_a, \ell_d\} = \{\ell_a\} \neq \emptyset$, so an edge is added between $\mathcal{S}_1$ and $\mathcal{S}_4$.

Given graphs created in this manner, the graph coloring and blocking analysis approaches presented in Sec. 3.1 and Sec. 3.4 can be applied.

4.3 Hierarchical Organization

Our initial approach to determining concurrency groups required constructing a graph with one vertex per request. Here we explore adding a layer of hierarchy to the request management scheme. We begin by considering a group of six requests: the five requests from Ex. 3.3 and one additional request.

Example 4.5. Consider the task set with the requests from Ex. 3.3 and a sixth request, $\mathcal{R}_6$, for $D_6 = \{\ell_a, \ell_e\}$ with a critical-section length of at most $L_6 = 55$ time units. Using the graph coloring approach described in Sec. 3.1, we can determine that four concurrency groups are required. One such grouping is shown in Fig. 9. This grouping results in worst-case blocking for all requests of $10 + 60 + 30 + 55 = 155$ time units.

This example highlights the impact a single request may have on the task system as a whole. Instead of the worst-case acquisition delay of 100 time units from Ex. 3.3, each request in this set may experience 155 time units of blocking.

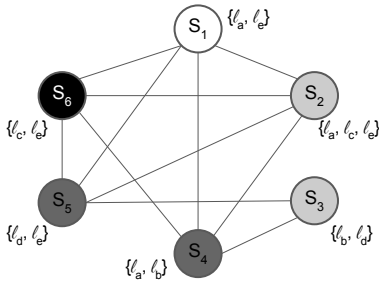


Figure 9: Four concurrency groups for requests $\mathcal{R}_1$ to $\mathcal{R}_6$.

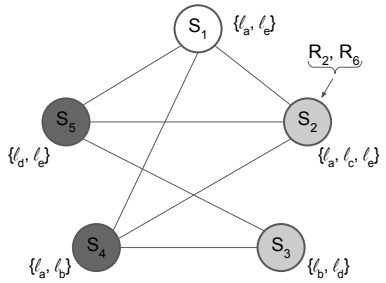


Figure 10: Three scheduling groups with one layer of hierarchy for $\mathcal{R}_2$ and $\mathcal{R}_6$.

We propose a slight modification to the creation of concurrency groups that will lower the worst-case blocking for most requests at the cost of increasing the worst-case blocking for a few requests. This modification allows a vertex to represent multiple requests and requires additional synchronization between those requests.

Example 4.5 (continued). For the purpose of this example, we choose $\mathcal{R}_2$ and $\mathcal{R}_6$ to be represented by a single vertex. (This choice reduces the required number of concurrency groups from four to three.) We construct a new graph with five vertices. We let each of $\mathcal{S}_1$, $\mathcal{S}_3$, $\mathcal{S}_4$, and $\mathcal{S}_5$ represent the request with the same index. $\mathcal{S}_2$ represents both $\mathcal{R}_2$ and $\mathcal{R}_6$.

When constructing a graph, edges are added between vertices based on all requests the vertices represent. Then, the graph coloring approach can be used as described in Sec. 3.1.

Example 4.5 (continued). When creating the graph shown in Fig. 10, edges from $\mathcal{S}_2$ are added based on requests that conflict with either $\mathcal{R}_2$ or $\mathcal{R}_6$.

To ease the discussion of when requests are satisfied under this scheme, instead of referring to the requests in a group, we will refer to their corresponding *slots*. Each vertex corresponds to exactly one slot, and multiple requests may compete to *occupy* the slot in the active phase of its group. The request that occupies its slot is satisfied according to the rules for requests above, and at most one request can occupy a slot at a time. To ensure this, additional coordination among requests that share a slot is required. Therefore, we require such requests to first acquire a mutex corresponding to the slot. This introduces an additional layer of hierarchy and additional blocking for these requests; a request must now wait until it acquires the mutex to occupy its slot and thus be eligible to be satisfied when its concurrency group becomes active. (Note that the use of a mutex in this context shifts the blocking a request experiences from one lock structure to another; this modification can be implemented without the use of a mutex for updating the overall lock state [18].)

The necessary coordination can be expressed as two additional rules:

G9 When multiple active requests compete for a slot, at most one request can occupy the slot at a time. Only requests occupying slots are considered in Rules G1 throug G8.

G10 When a request completes, it no longer occupies its slot. If another active request for the slot exists when a request completes, the transition between requests occupying the slot is atomic.

Theorem 4.3. *The worst-case acquisition delay a request $\mathcal{R}_i$ competing with s other requests to enter the concurrency group $\mathcal{G}_g$ is $(s+1)\sum_c L^{\mathcal{G}_c}_{max}$ if a FIFO-ordered mutex is used.*

Proof. If $\mathcal{R}_i$ is at the head of the queue for its slot, then it experiences acquisition delay of at most $\sum_c L^{\mathcal{G}_c}_{max}$ time units (Theorem 3.2).

Otherwise, if $\mathcal{R}_i$ is not at the head of the queue, there is a different request, $\mathcal{R}_j$, that occupies $\mathcal{R}_i$'s slot. $\mathcal{R}_j$ experiences acquisition delay of up to $\sum_c L^{\mathcal{G}_c}_{max}$ time units (Theorem 3.2). It then executes during $\mathcal{G}_g$'s active phase, which lasts for up to $L^{\mathcal{G}_g}_{max}$.

By Rule G10, when $\mathcal{R}_j$ completes, the subsequent request in the queue immediately occupies the slot in $\mathcal{G}_g$. Call this request $\mathcal{R}_x$. If another group is waiting, $\mathcal{R}_x$ is not satisfied immediately. Instead $\mathcal{G}_g$ transitions from the active state to the waiting state. Then, as in Theorems 3.1 and 3.2, $\mathcal{R}_x$ must wait for up to $k-1$ active phases of other groups incurring a delay of up to $\sum_{c\neq g} L^{\mathcal{G}_c}_{max}$ time units before being satisfied. If $\mathcal{R}_i \neq \mathcal{R}_x$, another active phase of $L^{\mathcal{G}_g}_{max}$ (during which $\mathcal{R}_x$ is satisfied) of duration up to $L^{\mathcal{G}_g}_{max}$ time units contributes to the acquisition delay experienced by $\mathcal{R}_i$. In this manner, for each of the $s-1$ requests between $\mathcal{R}_i$ and $\mathcal{R}_j$, $\mathcal{R}_i$ is delayed for an additional $\left(\sum_{c\neq g} L^{\mathcal{G}_c}_{max}\right) + L^{\mathcal{G}_g}_{max} = \sum_c L^{\mathcal{G}_c}_{max}$.

Finally, once $\mathcal{R}_i$ occupies its slot, it is delayed up to $\sum_{c\neq g} L^{\mathcal{G}_c}_{max}$ time units. Therefore, $\mathcal{R}_i$ experiences worst-case acquisition delay of $\sum_c L^{\mathcal{G}_c}_{max} + L^{\mathcal{G}_g}_{max} + (s-1)\sum_c L^{\mathcal{G}_c}_{max} + \sum_{c\neq g} L^{\mathcal{G}_c}_{max} = (s+1)\sum_c L^{\mathcal{G}_c}_{max}$ time units. □

Example 4.5 (continued). When the concurrency groups depicted in Fig. 10 are used, $\mathcal{R}_2$ and $\mathcal{R}_6$ have a maximum acquisition delay of $2\cdot 10+60+30 = 200$ time units. All other requests have maximum acquisition delay of 100 time units.

In essence, we can leverage additional knowledge about the tasks to increase blocking for some requests in order to lower blocking for other requests. The decision of which requests to map to the same vertex can depend on multiple factors. The resources that the requests require must be considered; for each request mapped to a slot, multiple edges may need to be added to the vertex. In general, to see the benefits of the tighter blocking bounds from

Protocol	Worst-Case Acquisition Delay	Total Overhead (μs)
CGLP	$\sum_g L_{max}^{G_g}$	3.1
U-C-RNLP	$C_i + 1 \cdot L_{max}$	13.0
G-C-RNLP	$C_i \cdot L_{max} + C_i \cdot L_i$	15.1
RNLP	$m \cdot L_i$	13.5
MCS	$m \cdot L_i$	0.7

Table 1: Blocking bounds and overhead of each protocol. In the bounds of C-RNLP, C_i is the number of requests which conflict with $\mathcal{R}_i$. (The reported overhead of the CGLP is the maximum of that measured with between two and ten concurrency groups.) Overhead values were measured on a dual-socket 8-cores-per-socket machine with 32 GB of DRAM, running Ubuntu 16.04.

Sec. 3.4, a vertex should represent requests with similar critical-section lengths. Furthermore, some tasks may be able to incur a higher amount of blocking and still meet their deadlines; this will depend on the execution time and period of each task.

An additional consideration is that in some applications it may be reasonable to expect requests for the same set of requests to have similar properties, such as critical-section length or ability to incur blocking. Grouping requests in this manner has the added benefit that a large number of identical requests only impacts the blocking of those requests and not other requests in the system.

5 EVALUATION

To evaluate the effectiveness of the CGLP, we compared it to prior real-time locking protocols in a schedulability study. Additionally, we explored how long it takes to determine concurrency groups.

5.1 Schedulability Study

Our primary method of evaluation is comparing schedulability of a variety of task sets when different synchronization protocols are used. The first protocol to which we compare the CGLP is the C-RNLP. The C-RNLP is the only existing protocol that solves the Transitive Blocking Chain Problem for nested write requests. There are two variants of the C-RNLP: the Uniform C-RNLP (U-C-RNLP) assumes uniform critical-section lengths and allows enqueuing based on this, while the General C-RNLP (G-C-RNLP) makes no such assumptions. We also compare the CGLP to the RNLP and to a simple group lock.[3] The blocking bounds and overhead of each protocol are summarized in Table 1.

We conducted our experiments using SchedCAT [1], an open-source real-time schedulability test toolkit. We used SchedCAT to randomly generate task systems, compute blocking bounds, and determine schedulability on a 16-core platform using G-EDF scheduling by Baruah's test [4]; we inflate the execution time of tasks based on the locking protocol overhead and blocking their requests may incur as described in [5]. In our experiments, we explored a broad space of task-system parameters, varying the individual task utilization, period, critical-section length, the percentage of tasks that issue requests, the probability that a given request is nested, and the number of resources requested for a nested request; named value sets are listed in Table 2, and the set of parameters used for our schedulability study are in Table 3. We define a *scenario* to be a setting of each of these parameters. Our study considered 384 scenarios; common trends are discussed here, and the full set of plots is available online along with the code [18]. For each scenario, at least 1,000 task systems were generated for every value of system utilization; we plot the percentage of these that are schedulable when no synchronization is required (NOLOCK) and when synchronization is provided by one of the five algorithms we compare. This evaluation took over 15 CPU-days of computation.

[3] We use a single MCS lock [15] to protect all resources for the group lock. Due to the scale of this study and the complexity of computing blocking bounds [26], we do not compare to a set of resource-ordered locks.

Category	Name	Value
Task Utilization	Medium-Light	[0.01,0.1]
	Medium	[0.1,0.4]
	Heavy	[0.5,0.9]
Critical-Section Length (μs)	Moderate	[15,100]
	Bimodal	[15,500] or [500,1000]
	Weighted Bimodal	[15,500] (prob: 0.7) or [500,1000] (prob: 0.3)
	Long	[100,1000]
Period (ms)	Short	[3,33]
	Long	[50,250]

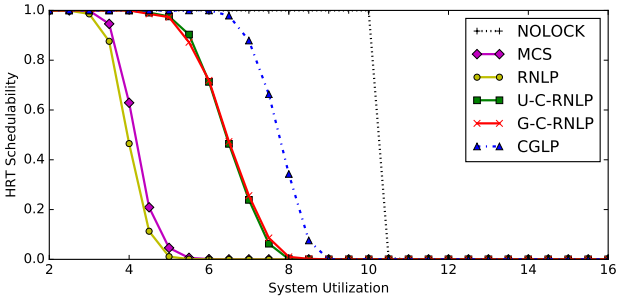
Table 2: Named parameter distributions. From each, a value is selected uniformly at random.

Category	Options
Task Utilization	Medium, Heavy
Period	Short, Long
Percentage Issuing Requests	50%, 80%, 100%
Critical-Section Length	Moderate, Bimodal, Weighted Bimodal, Long
Number of Resources	64
Nested Probability	0.1, 0.2, 0.5, 0.8
Nesting Depth	2, 4

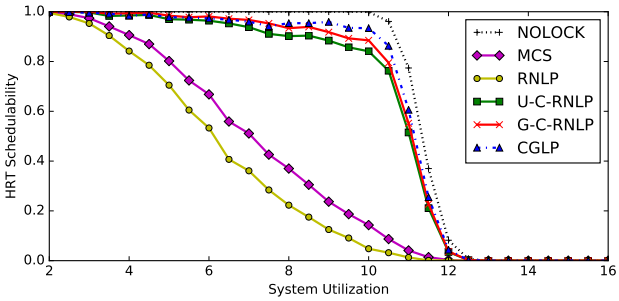
Table 3: Schedulability study parameter choices. Critical-section lengths are assigned with one of two methods: randomly for each request or within a range of the random length assigned to a group.

General results. For our initial schedulability study, we sought to separate the analysis of the CGLP from the process of determining the number of groups necessary for a single task system. To achieve this, we first compute the minimum number of groups necessary given the number of nested and non-nested requests in a given task system. For each request, we randomly choose a group and then selected the required number of resources from the group without replacement. If the distribution of requests is such that no group has sufficient resources when assigning resources to the next request, a new group is added. This method of request generation allows us to analyze the CGLP when the number of groups is small relative to the number of requests. We evaluate the average number of concurrency groups required for a given scenario in Sec. 5.2.

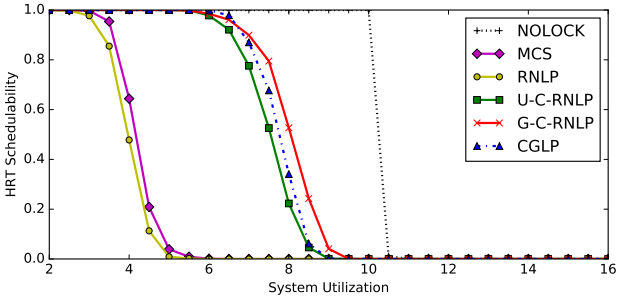
Fig. 11 shows the schedulability of task sets with varying utilizations in which 100% of the tasks issue requests. We chose these plots to represent some key trends we observed.



(a) Medium per-task utilization, nested probability 0.5

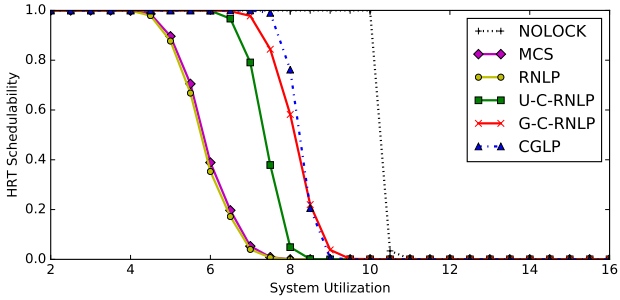


(b) Heavy per-task utilization, nested probability 0.5

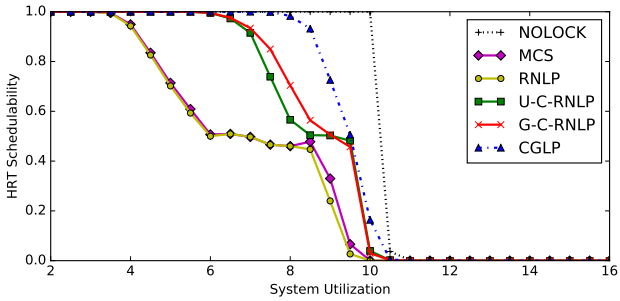


(c) Medium per-task utilization, nested probability 0.1

Figure 11: For these scenarios, the nesting depth was 4, critical-section lengths were moderate, periods were short and 100% of tasks issued requests.



(a) Randomly-chosen critical-section lengths



(b) Per-group critical-section lengths

Figure 12: For this scenario, task utilizations were medium, the nesting depth was 4, periods were long, critical sections were weighted bimodal, nested probability was 0.5, and 100% of tasks issued requests. Critical-section lengths were uniformly chosen from $[0.9, 1.1]$ times the group's value.

Obs. 1. *When many tasks issue requests and the nested probability is high, the CGLP tends to result in equal or higher schedulability than existing protocols.*

This observation is reflected for both medium (Fig. 11a) and heavy (Fig. 11b) task utilizations. In general, when 80% or 100% of tasks issue requests and those requests have probability of 0.5 or 0.8 of being nested for 4 resources, the CGLP is as good or better than existing approaches in 64.1% of scenarios. (The CGLP is always as good as or better than the RNLP or a group MCS lock.) When the probability of a request being nested is low, the G-C-RNLP tends to outperform all other protocols. This trend is reflected for medium utilization tasks with a lower nested request probability in Fig. 11c.

Critical-section length considerations. In Sec. 4.1, we explored the benefits of creating concurrency groups with requests of similar critical-section lengths. To simulate this scenario in the context of our schedulability study, we assign a critical-section length for each group we generate. The requests generated that belong to a given concurrency group are assigned a critical-section length randomly chosen from a uniform distribution of $[0.9, 1.1]$ times the group's pre-assigned critical-section length. The benefits of grouping tasks by critical-section length are captured in Fig. 12.

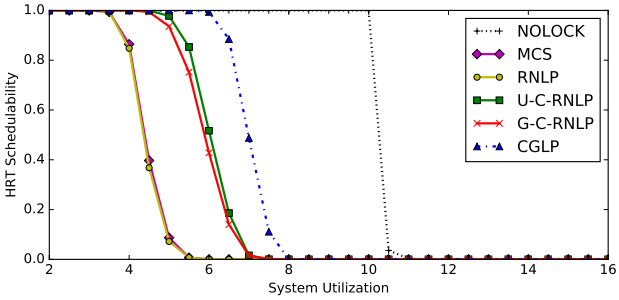
Obs. 2. *When the tasks are grouped by critical-section length, the schedulability of the CGLP increases.*

This trend is shown in Fig. 12a and Fig. 12b and is as expected for the CGLP; the bounds in Sec. 3.4 capture the benefit of grouping requests by similar critical-section length. In the online appendix [18], we discuss analogous methods by which we tightened the computed bounds for the C-RNLP variants.

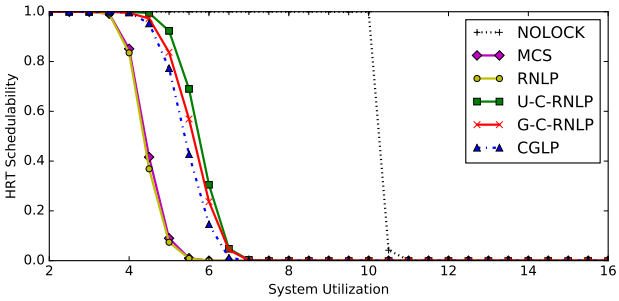
Obs. 3. *The change in the distribution of critical-section lengths significantly impacts the schedulability of existing protocols.*

This is illustrated in Fig. 12b, which shows an inflection point on each of these curves. One method by which we tightened bounds accounts for the largest critical-section lengths; changing this distribution has a significant impact on the computed blocking, in general increasing schedulability for existing protocols.

Importance of few concurrency groups. The blocking bounds of the CGLP presented in Sec. 3.4 depend heavily on the number of



(a) Minimum of 4 groups



(b) Minimum of 10 groups

Figure 13: For this scenario, task utilizations were medium, the nesting depth was 4, periods were long, critical sections were long, nested probability was 80%, and 100% of tasks issued requests.

concurrency groups. For some task sets the distribution of requests over the set of resources may result only a few groups relative to the number of requests. When this is not the case, a method like that described in Sec. 4.3 must be explored.

For this next component of our study, we modify how we choose the number of groups. Here, we specify the number of groups from which requests may be chosen. No modifications are made after the requests are chosen, so this selection determines the number of concurrency groups. The impact of the number of concurrency groups is depicted in Fig. 13.

Obs. 4. *Schedulability under the CGLP decreases as the number of groups increases.*

This is depicted in Fig. 13, as expected; based on the bound given in Sec. 3.4, adding a group adds an additional critical-section to the computation of worst-case blocking. This decrease in schedulability emphasizes the importance of using the minimum number of concurrency groups.

5.2 Determining the Concurrency Groups

In Sec. 3.1, we described how the offline partitioning of requests into concurrency groups can be cast as a Vertex Coloring Problem. We encode this problem as an Integer Linear Program (ILP) by using binary variables to indicate a color assignment for each vertex [19].

To test how long it takes to determine concurrency groups for a given task set, we generated random requests for a task set in which tasks had long periods, every task issued a request, and nested requests required four resources. We then used an ILP solver [11] to determine the minimum number of groups required for varying probabilities of nested requests, as shown in Fig. 14. For each scenario, we generated 50 task sets.

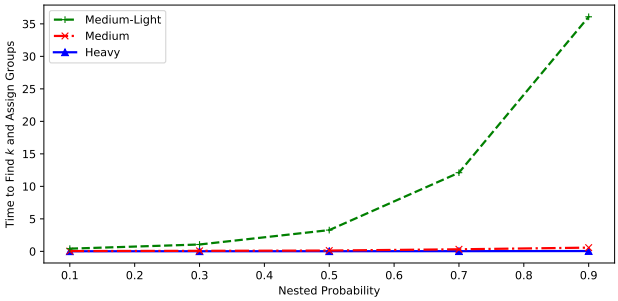


Figure 14: Average time to compute the minimum coloring per nested probability. Error bars indicate minimum and maximum values. Each data point represents 50 random task sets.

Task Utilization	Average Number of Requests	Average k	Time to Find k, Assign Groups
Heavy	23.4	3.7	0.026 s
Medium	64.7	6.7	0.227 s
Medium-Light	291.6	19.8	10.59 s

Table 4: Average time required to find minimum group number (and an assignment of requests to groups).

Obs. 5. *While the connection of the problem of determining groups to the NP-complete Vertex Coloring Problem may seem like a serious liability, the ILP solver was almost always able to quickly find such groups across a wide spectrum of scenarios.*

The average time in which the ILP solver determined the minimum number of groups and assigned these groups is shown in Fig. 14 and Table 4.

6 CONCLUSION

In this paper, we have presented the CGLP, a nested real-time locking protocol that solves both the Transitive Blocking Chain Problem and the Request Timing Problem. The CGLP determines concurrency groups offline to reduce the blocking experienced by requests. We provided the worst-case acquisition delay and showed how the CGLP can be improved by considering critical-section lengths. Additionally, we presented an extension that allows worst-case blocking to be tuned based on task parameters.

We evaluated the CGLP on the basis of schedulability and showed that for task system with mostly nested write requests, it tends to outperform existing protocols. We also showed that, for many tasks systems, the minimum number of concurrency groups can be computed by an ILP solver very quickly.

As future work, we plan to implement a mechanism that sorts requests into groups based on how much blocking a given request can incur and remain schedulable. Additionally, we will explore the benefits of handling mixed-type requests. Finally, we will incorporate this as a component of a larger protocol by merging work that handles non-nested requests and read requests efficiently with the CGLP, which efficiently manages nested write requests.

REFERENCES

[1] 2019. SchedCAT: Schedulability test collection and toolkit. https://github.com/brandenburg/schedcat. Accessed: 2019-02-07.

[2] Tobias Bandh, Georg Carle, and Henning Sanneck. 2009. Graph coloring based physical-cell-ID assignment for LTE networks. In *IWCMC '09*.

[3] N. Barnier and P. Brisset. 2004. Graph coloring for air traffic flow management. *Annals of operations research* 130, 1-4 (2004).

[4] S. Baruah. 2007. Techniques for multiprocessor global schedulability analysis. In *RTSS '07*.

[5] B. Brandenburg. 2011. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. Ph.D. Dissertation. University of North Carolina, Chapel Hill, NC.

[6] A. Burns and A. Wellings. 2013. A schedulability compatible multiprocessor resource sharing protocol - MrsP. In *ECRTS '13*.

[7] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. 1981. Register allocation via coloring. *Computer languages* 6, 1 (1981).

[8] D. Faggioli, G. Lipari, and T. Cucinotta. 2010. The multiprocessor bandwidth inheritance protocol. In *ECRTS '10*.

[9] D. Faggioli, G. Lipari, and T. Cucinotta. 2012. Analysis and implementation of the multiprocessor bandwidth inheritance protocol. *Real-Time Systems* 48, 6 (2012).

[10] J. Garrido, S. Zhao, A. Burns, and A. Wellings. 2017. Supporting nested resources in MrsP. In *Ada-Europe International Conference on Reliable Software Technologies '17*.

[11] LLC Gurobi Optimization. 2018. Gurobi Optimizer Reference Manual. http://www.gurobi.com

[12] J. Havender. 1968. Avoiding deadlock in multitasking systems. *IBM systems journal* 7, 2 (1968).

[13] C. Jarrett, B. Ward, and J. Anderson. 2015. A Contention-Sensitive Fine-Grained Locking Protocol for Multiprocessor Real-Time Systems. In *RTNS '15*.

[14] D. Marx. 2004. Graph colouring problems and their applications in scheduling. *Periodica Polytechnica Electrical Engineering* 48, 1-2 (2004).

[15] J. Mellor-Crummey and M. Scott. 1991. Algorithms for scalable synchronization of shared-memory multiprocessors. *Transactions on Computer Systems* 9, 1 (1991).

[16] C. Nemitz, T. Amert, and J. Anderson. 2018. Using Lock Servers to Scale Real-Time Locking Protocols: Chasing Ever-Increasing Core Counts. In *ECRTS '18*.

[17] C. Nemitz, T. Amert, and J. Anderson. 2019. Real-time multiprocessor locks with nesting: optimizing the common case. *Real-Time Systems* 55, 2 (2019).

[18] C. Nemitz, T. Amert, M. Goyal, and J. Anderson. 2019. Concurrency Groups: A New Way to Look at Real-Time Multiprocessor Lock Nesting (extended version). http://www.cs.unc.edu/Ëœanderson/papers.html

[19] S. Palladino. 2010. Modelling graph coloring with integer linear programming. https://manas.tech/blog/2010/09/16/modelling-graph-coloring-with-integer-linear-programming.html.

[20] R. Rajkumar, L. Sha, and J. Lehoczky. 1988. Real-Time Synchronization Protocols for Multiprocessors. In *RTSS '88*.

[21] H. Takada and K. Sakamura. 1995. Real-time scalability of nested spin locks. In *RTCSA '95*.

[22] B. Ward. 2016. *Sharing Non-Processor Resources in Multiprocessor Real-Time Systems*. Ph.D. Dissertation. University of North Carolina, Chapel Hill, NC.

[23] B. Ward and J. Anderson. 2012. Supporting nested locking in multiprocessor real-time systems. In *ECRTS '12*.

[24] B. Ward and J. Anderson. 2013. Fine-grained multiprocessor real-time locking with improved blocking. In *RTNS '13*.

[25] B. Ward and J. Anderson. 2014. Multi-resource real-time reader/writer locks for multiprocessors. In *IPDPS '14*.

[26] A. Wieder and B. Brandenburg. 2014. On the complexity of worst-case blocking analysis of nested critical sections. In *RTSS '14*.

[27] S. Zhao, J. Garrido, A. Burns, and A. Wellings. 2017. New schedulability analysis for MrsP. In *RTCSA '17*.