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

Catherine E. Nemitz · Tanya Amert · Manish Goyal · James H. Anderson

Abstract When designing a real-time multiprocessor locking protocol, the allowance of lock nesting creates complications that can inhibit 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 these benefits, as well as an approach to determining such groups using an Integer Linear Program (ILP) solver, which we show to be efficient in practice.

Keywords multiprocess locking protocols · nested locks · priority-inversion blocking · real-time locking protocols

1 Introduction

Although real-time multiprocessor locking protocols have been studied for over thirty years [48], the issue of enabling unrestricted *lock nesting*—*i.e.*, a task

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.

The University of North Carolina at Chapel Hill
E-mail: {nemitz, tamert, manishg, anderson}@cs.unc.edu

holding locks on several resources simultaneously—in an efficient manner was considered only relatively recently [54]. 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 [1,8,16]. 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 access 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 Consider a scenario with four tasks and five resources, ℓ_a through ℓ_e. 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 been issued and enqueued in task-index order. The maximum duration of each request is illustrated by a box of that height. 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.

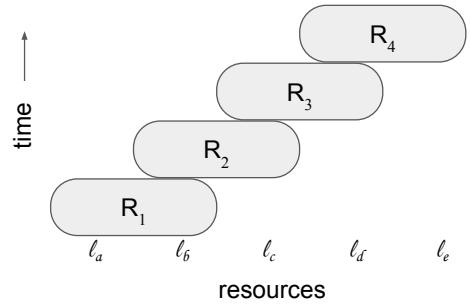


Fig. 1 FIFO-ordering.

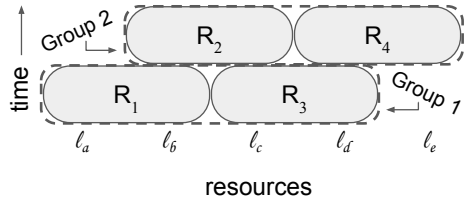


Fig. 2 Optimized offline ordering.

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 (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 [35, 42], worst-case blocking under these protocols 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. These gaps inhibit parallel execution.

Example 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 and 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

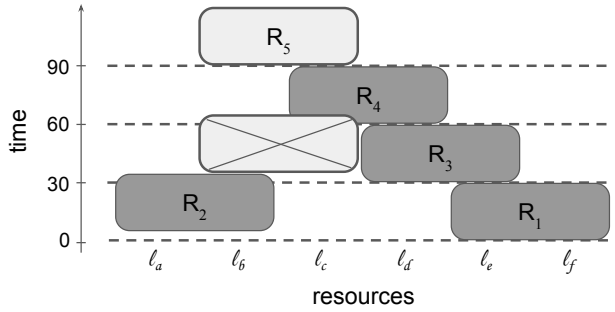


Fig. 3 An illustration of the Request Timing Problem. $\mathcal{R}_5$ may not be inserted in the earlier slot marked by an 'X', as this would delay an already issued request.

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 predetermine the "slots" into which requests are inserted. Because this determination is made offline, it is not subject to runtime timing variations.

In many protocols, having to deal with requests of different durations can also cause "gaps" similar to that in Ex. 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. A preliminary version of this work framed the construction of concurrency groups as a graph-coloring problem and presented a method for forming these groups with an Integer Linear Program (ILP) [43]. In this work, we present that ILP in full. We then explore approaches for determining groups that improve worst-case blocking bounds and present an alternate ILP that leverages additional task system information to improve blocking bounds. To assess the CGLP, we analyze the offline component by examining its execution time and the online component by presenting the results of a schedulability study.

Organization. We begin with required 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 Secs. 4–6. In Sec. 4, we present an ILP that minimizes blocking instead of

the number of concurrency groups. In Sec. 5, we extend the previous two ILPs to account for mixed-type requests. In Sec. 6, we explore incorporating additional task system parameters into the optimization problem. We present the aforementioned schedulability study in Sec. 7, discuss related work in Sec. 8, and conclude in Sec. 9.

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 of n implicit-deadline tasks $\{\tau_1, ..., \tau_n\}$ on a multiprocessor platform with m processors. A task τ_i is further specified by its worst-case execution time C_i and its minimum job separation T_i. 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* while it *holds* all of its required resources, denoted $\mathcal{D}_i$.[1] $\mathcal{R}_i$ executes its critical section non-preemptively for at most L_i time units before it *completes* and *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 $\mathcal{D}_i$ or a *read* request if other requests may access $\mathcal{D}_i$ concurrently. A request is a *mixed* request if it requires mutually exclusive access to some resources in $\mathcal{D}_i$ and other requests may concurrently access the remaining resources in $\mathcal{D}_i$.

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 [42]. We also consider how our protocol can be extended to accommodate mixed 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 non-preemptively until its request is satisfied.

[1] We assume the use of dynamic group locks [54], which coalesce all resources a task may require concurrently under a single request. For example, if a task requires access to ℓ_a and then conditionally requires access to either ℓ_b or ℓ_c, it issues a single request for $\{\ell_a, \ell_b, \ell_c\}$.

Existing protocols. In this work we compare our new protocol to four existing protocols. The broader context of these protocols is discussed in Sec. 8, but we briefly describe these protocols here.

First, we compare to a single MCS lock [38] used to protect all resources as a static group lock. The MCS lock is a FIFO-ordered lock with very low overhead, but its application as a single group lock severely limits parallelism. The remaining three protocols to which we compare are in the *real-time nested locking protocol (RNLP)* family of protocols [35, 52–55]. Each of these protocols also use dynamic group locking [54] to coalesce each set of nested requests into a single request for multiple resources. (Other approaches to handling nested resource access in real-time systems are discussed in Sec. 8.) We compare to the basic RNLP [53], which provides asymptotically optimal pi-blocking for nested requests. It does so by using per-resource queues that are timestamp-ordered based on the time each request was issued. We also compare to the two C-RNLP variants (the U-C-RNLP and the G-C-RNLP) [35], which can provide a tighter blocking bound that is based on the number of requests with which the request of interest shares resources. The C-RNLP also uses per-resource queues, but allows later-issued requests to cut ahead of enqueued requests when doing so will not increase blocking for these requests. This more complex protocol necessarily has higher overhead in order to achieve lower blocking.

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 (we will extend our consideration to mixed-type requests in Sec. 5). Then we show how phase-based access to resources can be achieved by generalizing prior protocols that orchestrate phases of only two or three types [41, 42]. We finish this section by bounding the worst-case blocking any request may incur under the CGLP.

3.1 Offline Group Creation via Graph Coloring

A k-coloring of a graph is an *injective mapping* of its vertices to a set of colors, K, such that $|K| = k$. A coloring is *proper* if no two adjacent vertices

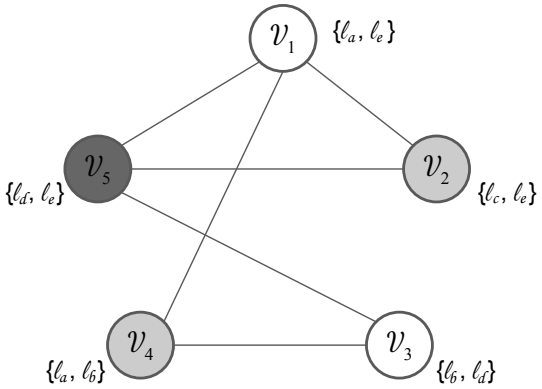


Fig. 4 An example coloring.

are assigned the same color. A graph is k-colorable if it has a proper k-coloring. The Vertex Coloring Problem (VCP) for a graph entails finding the *chromatic number*, defined as the smallest integer k for which the graph is 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 VCP in two steps. First, for each request $\mathcal{R}_i$, we create a corresponding vertex $\mathcal{V}_i$. Once we have added all vertices to the graph, we add edges. An edge is added between $\mathcal{V}_i$ and $\mathcal{V}_j$, where $i \neq j$, if $\mathcal{D}_i \cap \mathcal{D}_j \neq \emptyset$.

Example 3 Consider a task set that produces five requests: $\mathcal{R}_1$ for $\mathcal{D}_1 = \{\ell_a, \ell_e\}$, $\mathcal{R}_2$ for $\mathcal{D}_2 = \{\ell_c, \ell_e\}$, $\mathcal{R}_3$ for $\mathcal{D}_3 = \{\ell_b, \ell_d\}$, $\mathcal{R}_4$ for $\mathcal{D}_4 = \{\ell_a, \ell_b\}$, and $\mathcal{R}_5$ for $\mathcal{D}_5 = \{\ell_d, \ell_e\}$. The graph representation of these requests is shown in Fig. 4. For example, $\mathcal{V}_4$ is connected to $\mathcal{V}_1$ and $\mathcal{V}_3$ because $\mathcal{D}_4 \cap \mathcal{D}_1 = \{\ell_a\}$ and $\mathcal{D}_4 \cap \mathcal{D}_3 = \{\ell_b\}$. $\mathcal{V}_4$ does not have an edge to either $\mathcal{V}_2$ or $\mathcal{V}_5$, as $\mathcal{D}_4 \cap \mathcal{D}_2 = \emptyset$ and $\mathcal{D}_4 \cap \mathcal{D}_5 = \emptyset$.

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 $\mathcal{V}_i$ is assigned Color g, then $\mathcal{R}_i \in \mathcal{G}_g$.

Example 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 VCP, none of the requests in a given concurrency group require any overlapping resources.

Theorem 1 *All requests in a given concurrency group $\mathcal{G}_g$ created via a solution to the corresponding VCP may be satisfied concurrently (i.e., mutual exclusion will not be violated).*

Proof Suppose not. Therefore, there exist two requests $\mathcal{R}_i$ and $\mathcal{R}_j$ in the same concurrency group $\mathcal{G}_g$ that share a resource (*i.e.*, $\mathcal{D}_i \cap \mathcal{D}_j \neq \emptyset$). By the method of constructing the VCP described above, there is an edge between $\mathcal{V}_i$ and $\mathcal{V}_j$. Thus, $\mathcal{V}_i$ and $\mathcal{V}_j$ could not both be assigned Color g as a valid solution to the problem. Therefore, $\mathcal{R}_i$ and $\mathcal{R}_j$ cannot both be in $\mathcal{G}_g$. Contradiction. □

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.

3.2 Implementation of Offline Component

We encode an instance of the VCP as an Integer Linear Program (ILP) by using binary variables to indicate a color assignment for each vertex. The following formulation of this problem is based on the description in prior work [45].

Variables. We use the binary variable $x_{i,g}$ to indicate whether $\mathcal{V}_i$ is assigned Color g (*i.e.*, $\mathcal{R}_i$ belongs to $\mathcal{G}_g$). We let the binary variable $color_g$ denote whether Color g is used to color any vertex.

Constraints. The first constraint of our ILP enforces that each vertex is assigned exactly one color.

Constraint 1 $\forall i : \sum_c x_{i,c} = 1$

By using binary variables, we have $x_{i,c} \in \{0, 1\}$. Thus, summing $x_{i,c}$ across all colors for a given vertex $\mathcal{V}_i$ yields the number of colors that the vertex has been assigned.

Our second constraint enforces that adjacent vertices may not be assigned the same color.

Constraint 2 $\forall c \; \forall \mathcal{R}_i, \mathcal{R}_j : \mathcal{R}_i \neq \mathcal{R}_j \wedge \mathcal{D}_i \cap \mathcal{D}_j \neq \emptyset : x_{i,c} + x_{j,c} \leq 1$

When considering adjacent vertices $\mathcal{V}_i$ and $\mathcal{V}_j$, for any color, at most one of the vertices may be assigned that color.

Our third constraint captures whether a given color has been used.

Constraint 3 $\forall i \; \forall c : color_c \geq x_{i,c}$

If any vertex is assigned Color c, $color_c$ will be 1.

Finally, our objective function reflects our goal of minimizing the number of colors used.

Objective $min \sum_c color_c$

To describe this problem as an ILP, we must include a variable for each available color. This requires us to pre-determine a sufficient number of colors. Two simple methods can be used to obtain a maximum number of colors: (1) the maximum is bounded by the number of vertices, or (2) the maximum may be more tightly bounded by applying a greedy coloring algorithm (described in Sec. 7.1).

We now specify the ILP corresponding to the running example from above.

Example 3 (continued) Let us suppose we have applied a greedy coloring approach that yielded four colors. Thus, we know that the minimum k is at most four. For each request, Constraint 1 yields one equality, resulting in five total. Constraint 2 results in inequalities for each color: for a given color, there is one inequality per edge in the graph. This results in 24 inequalities. Finally, Constraint 3 results in 20 inequalities, five per color, to enforce that each $color_g$ variable accurately captures whether Color g has been assigned to any vertex. This yields the following ILP.

$$\min_{c} \quad \sum_{c=1}^{4} color_c$$

$$\text{s.t.} \quad x_{1,1} + x_{1,2} + x_{1,3} + x_{1,4} = 1$$

repeat above equality for Vertices 2-5

$$x_{1,1} + x_{2,1} \leq 1$$
$$x_{1,1} + x_{4,1} \leq 1$$
$$x_{1,1} + x_{5,1} \leq 1$$
$$x_{2,1} + x_{5,1} \leq 1$$
$$x_{3,1} + x_{4,1} \leq 1$$
$$x_{3,1} + x_{5,1} \leq 1$$

repeat above six inequalities for Colors 2-4

$$color_1 \geq x_{1,1}$$
$$color_1 \geq x_{2,1}$$
$$color_1 \geq x_{3,1}$$
$$color_1 \geq x_{4,1}$$
$$color_1 \geq x_{5,1}$$

repeat above five inequalities for Colors 2-4

Though the VCP is NP-hard, we show in Sec. 7 that, for many systems, groups can be determined in a reasonable amount of time (in our experiments, using the Gurobi ILP solver [33]). What remains is to coordinate access to these groups of requests during runtime.

3.3 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 [41] and 3-phased [42] reader-reader locking protocols.

Example 3 (continued) No synchronization protection is required between requests $\mathcal{R}_1$ and $\mathcal{R}_3$, both in $\mathcal{G}_1$, as they do not shared resources. However, $\mathcal{G}_1$ and $\mathcal{G}_2$ cannot be allowed to execute concurrently.

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 (continued) As depicted in Fig. 5, $\mathcal{R}_1$ is issued at time $t = 10$. Because no other groups are active at $t = 10$, $\mathcal{G}_1$ becomes active immediately, by Rule G2. By Rule G4, $\mathcal{R}_1$ is satisfied immediately. At $t = 15$, $\mathcal{R}_5$ 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 $t = 60$.

- **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 (continued) $\mathcal{G}_3$ is active from $t = 60$ to $t = 110$. $\mathcal{R}_5$ 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.

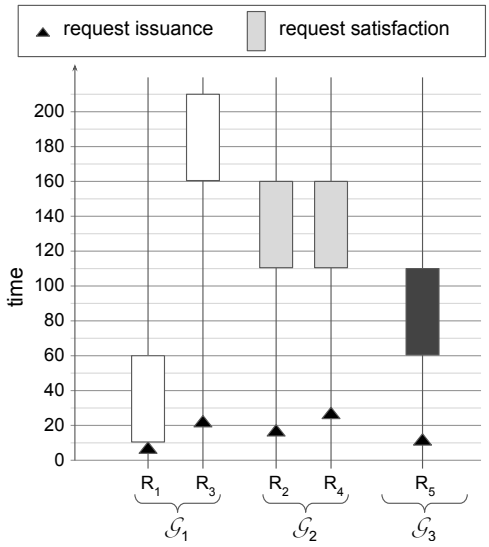


Fig. 5 Trace of executions of requests in Ex. 3.

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 (continued) $\mathcal{R}_3$ is issued at time $t = 25$, 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. (If $\mathcal{R}_3$ were instead satisfied immediately, the current phase of $\mathcal{G}_1$ would not end until time $t = 75$, delaying the satisfaction of $\mathcal{R}_5$ by 15 time units; such delays could lead to starvation of waiting groups.)

The above rules capture how the k concurrency groups alternate between active phases. These, along with the non-preemptive execution of critical sections, prevent deadlock. We next discuss our spin-based implementation of the R^{k}LP.

3.4 Implementation of Online Component

The R^{k}LP builds on the reader-reader locking protocol [41] and the R^{3}LP [42]. Here we describe the key components of the R^{k}LP implementation broadly; it is very similar to that of the R^{3}LP.

For each group, a set of counters is maintained. A newly issued request is assigned the current value of the counter that tracks how many requests have been issued. This counter, along with two others, serves to identify how many requests are active and distinguish which requests in the group are satisfied and which are waiting.

The R^{k}LP can be implemented without a mutex by instead using a standard atomic read and update mechanism on a shared bit vector. Two bits per concurrency group are maintained in the shared bit vector; one bit indicates that a request in the group is active, and the other denotes the phase of that group (to prevent a race condition in which a request from a different group fails to read the bit vector between phases of this group). Based on this construction, a 64-bit vector allows for 32 groups, or using a double-width compare-and-swap mechanism allows for 64 groups. While this may limit some applications, if the number of groups is larger than the number of processors, minimal analytical advantage can be gained by forming concurrency groups. Thus, this constraint (at most 64 groups being supported without the use of a mutex) is primarily a concern for systems with more than 64 processors.

3.5 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 establish a bound on the worst-case acquisition delay.

Lemma 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 2 *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 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 (continued) When $\mathcal{R}_3$ is issued at $t = 25$ 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.

3.6 Refining the Blocking Bound

Up to this point, we have not specified the critical-section lengths, so we treated each as L_{max}. When requests have varying critical-section lengths, the bound in Theorem 2 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 4 Here, we use the same set of requests from Ex. 3, but instead 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 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 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, which occurs within $L_{max}^{\mathcal{G}_g}$ time units. □

Theorem 3 *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 2, $\mathcal{R}_i$ 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 4 (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, 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 2.

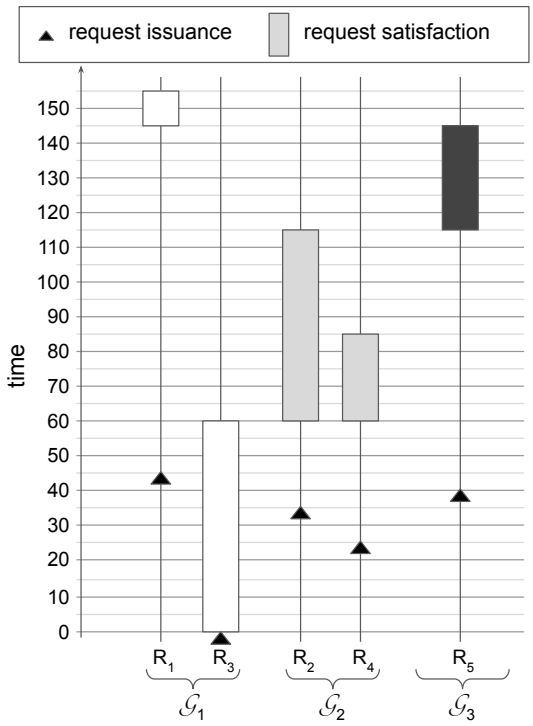


Fig. 6 An illustration of the maximum blocking for $\mathcal{R}_1$ in Ex. 4.

4 Alternate Coloring Choices

Now that we have explained the fundamental components of the CGLP, we discuss several extensions to the protocol. In this section, we focus on the benefits of allowing critical-section lengths to factor into the group assignments. This alternative method minimizes the total blocking experienced by all tasks. Therefore, when comparing protocols on the basis of schedulability, we expect this variant to outperform the basic CGLP from Sec. 3; the results of this comparison are presented in Sec. 7 (Obs. 5).

4.1 Motivation

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. We use the following examples to motivate a different method of grouping requests.

Example 4 (continued) Continuing the running example from the prior section, there are multiple ways of forming three concurrency groups for this set

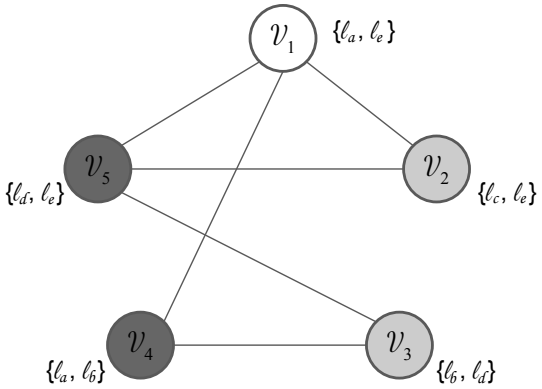


Fig. 7 An alternate coloring.

$\mathcal{G}_1$	$\mathcal{G}_2$	$\mathcal{G}_3$	$\mathcal{G}_4$	$\sum_{c=1}^{4} L_{max}^{\mathcal{G}_c}$
$\mathcal{R}_1$	$\mathcal{R}_2$	$\mathcal{R}_3$	$\mathcal{R}_4$	110
$\mathcal{R}_1$	$\mathcal{R}_2$, $\mathcal{R}_4$	$\mathcal{R}_3$	–	100
$\mathcal{R}_1$, $\mathcal{R}_4$	$\mathcal{R}_2$	$\mathcal{R}_3$	–	80
$\mathcal{R}_1$, $\mathcal{R}_3$	$\mathcal{R}_2$	$\mathcal{R}_4$	–	100
$\mathcal{R}_1$, $\mathcal{R}_3$	$\mathcal{R}_2$ $\mathcal{R}_4$	–	–	90

Table 1 Five possible groupings of the requests from Ex. 5 with the R^{k}LP blocking bounds computed. Using the minimum of two groups does not result in the lowest blocking.

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 could 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 when assigning groups.

Example 4 (continued) By Theorem 3, 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. 4 highlights the improvements in worst-case blocking that can be achieved by creating concurrency groups based on the critical-section lengths of the requests. In fact, taking minimizing blocking as our primary goal may require more than k groups. We illustrate this with an example.

Example 5 Consider the following set of requests: $\mathcal{R}_1$ with $\mathcal{D}_1 = \{\ell_a, \ell_b\}$, $\mathcal{R}_2$ with $\mathcal{D}_2 = \{\ell_b, \ell_c\}$, $\mathcal{R}_3$ with $\mathcal{D}_3 = \{\ell_c, \ell_d\}$, and $\mathcal{R}_4$ with $\mathcal{D}_4 = \{\ell_d, \ell_e\}$. Here, $L_1 = 60$, $L_2 = 10$, $L_3 = 10$, and $L_4 = 30$ time units. In Table 1, we show all possible groupings of these requests; those with different group number

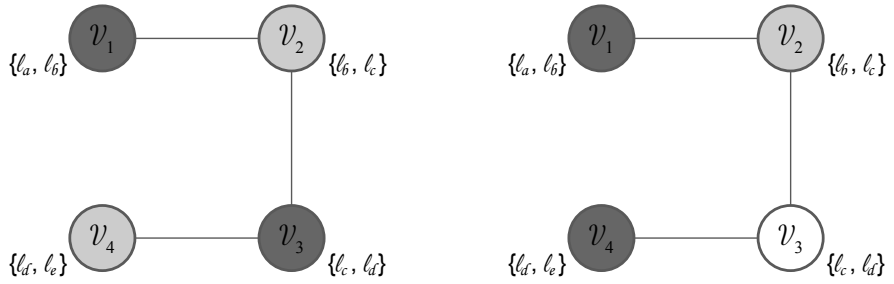


Fig. 8 For the requests in Ex. 5, the corresponding minimum coloring is on the left, and the coloring that achieves the minimum blocking is on the right.

assignments are simply permutations of these groups and result in the same summed blocking.

Ex. 5 illustrates that the minimum coloring does not always result in the lowest blocking. For this task set, the lowest blocking is found by using three groups to yield a blocking bound of 80 time units instead of the bound of 90 time units produced when only two groups are used. These choices of colorings are depicted in Fig. 8.

4.2 Minimizing Blocking

To minimize the impacts of synchronization on the system, we seek to minimize blocking with our assignment of requests to concurrency groups. We do so by developing an ILP that follows many of the same principles as the ILP presented in Sec. 3.2. We begin by describing the variables we use and then detail the constraints we apply.

Variables. As in Sec. 3.2, we use the notion of graph coloring to guide our approach. We use the binary variable $x_{i,g}$ to indicate whether $\mathcal{V}_i$ is assigned Color g (*i.e.*, $\mathcal{R}_i$ belongs to $\mathcal{G}_g$). To capture $L_{max}^{\mathcal{G}_g}$, we use the variable $duration_g$. Here, we no longer minimize the number of colors, as described below. Thus, the number of colors in the ILP is given by the number of vertices in the graph.

Constraints. We now present our ILP to determine groups while minimizing blocking. We use Constraints 1 and 2 from Sec. 3.2, which are restated below.

Constraint 1 $\forall i : \sum_c x_{i,c} = 1$

Constraint 2 $\forall c\ \forall \mathcal{R}_i, \mathcal{R}_j : \mathcal{R}_i \neq \mathcal{R}_j \wedge \mathcal{D}_i \cap \mathcal{D}_j \neq \emptyset : x_{i,c} + x_{j,c} \leq 1$

We add a constraint that forces a given $duration_g$ to capture the largest critical-section length of the requests in $\mathcal{G}_g$.

Constraint 3 $\forall i\ \forall c : duration_c \geq x_{i,c} \cdot L_i$

Intuitively, if $\mathcal{R}_i$ is in $\mathcal{G}_g$, then $L_{max}^{\mathcal{G}_g}$ must be at least L_i. When the ILP is formed, the critical-section lengths are incorporated into the model. Note that in the context of a given task system these are constants.

Our objective function is to minimize $\sum duration_c$ over all possible colors. This minimizes overall blocking, as computed by the expression in Theorem 3.

Objective $min \sum_c duration_c$

We illustrate how this is used with the example task system from above.

Example 5 (continued) The ILP corresponding to this set of four requests is listed below. Constraint 1 results in four equalities, Constraint 2 results in twelve inequalities, and Constraint 3 results in sixteen inequalities.

$$\min \quad \sum_{c=1}^{4} duration_c$$

$$\text{s.t.} \quad x_{1,1} + x_{1,2} + x_{1,3} + x_{1,4} = 1$$

repeat above equality for Vertices 2-4

$$x_{1,1} + x_{2,1} \leq 1$$
$$x_{2,1} + x_{3,1} \leq 1$$
$$x_{3,1} + x_{4,1} \leq 1$$

repeat above three inequalities for Colors 2-4

$$duration_1 \geq x_{1,1} \cdot 60$$
$$duration_1 \geq x_{2,1} \cdot 10$$
$$duration_1 \geq x_{3,1} \cdot 10$$
$$duration_1 \geq x_{4,1} \cdot 30$$

repeat above four inequalities for Colors 2-4

5 Mixed-Type Requests

Recall that 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 value(s) from 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.

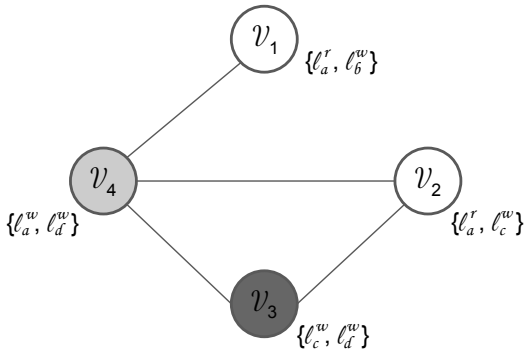


Fig. 9 Graph of mixed-type requests.

5.1 Graph Creation

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 $\mathcal{D}_i$ required by a request $\mathcal{R}_i$, we denote the type of access required (read or write) with a superscript. For example, $\mathcal{D}_i = \{\ell_a^r, \ell_b^w\}$ indicates that $\mathcal{R}_i$ requires read access to ℓ_a and write access to ℓ_b.

Example 6 Consider a set of requests, $\mathcal{R}_1$ through $\mathcal{R}_4$, which require resources $\mathcal{D}_1 = \{\ell_a^r, \ell_b^w\}$, $\mathcal{D}_2 = \{\ell_a^r, \ell_c^w\}$, $\mathcal{D}_3 = \{\ell_c^w, \ell_d^w\}$, and $\mathcal{D}_4 = \{\ell_a^w, \ell_d^w\}$. 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 $\mathcal{D}_i^w = \{\ell_y | \ell_y^w \in \mathcal{D}_i\}$ 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 $(\mathcal{D}_i^w \cap \mathcal{D}_j \neq \emptyset) \vee (\mathcal{D}_i \cap \mathcal{D}_j^w \neq \emptyset)$.

Example 6 (continued) The graph corresponding to this set of requests is shown in Fig. 9. Here, $\mathcal{D}_1^w = \{\ell_b\}$. 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 $(\mathcal{D}_1^w \cap \mathcal{D}_2) = (\{\ell_b\} \cap \{\ell_a, \ell_c\}) = \emptyset$ and $(\mathcal{D}_1 \cap \mathcal{D}_2^w) = (\{\ell_a, \ell_b\} \cap \{\ell_c\}) = \emptyset$, so no edge is added between $\mathcal{V}_1$ and $\mathcal{V}_2$. This fits the intuition that $\mathcal{R}_1$ and $\mathcal{R}_2$ could be satisfied concurrently. For $\mathcal{R}_1$ and $\mathcal{R}_4$, $(\mathcal{D}_1 \cap \mathcal{D}_4^w) = (\{\ell_a, \ell_b\} \cap \{\ell_a, \ell_d\}) = \{\ell_a\} \neq \emptyset)$, so an edge is added between $\mathcal{V}_1$ and $\mathcal{V}_4$.

Given graphs created in this manner, the blocking analysis presented in Sec. 3.6 can be applied.

5.2 Modifications to ILP

Both of the ILPs we presented previously can be modified to account for mixed requests (or read requests) by replacing Constraint 2 with the following:

Constraint 2 $\forall c \; \forall \mathcal{R}_i, \mathcal{R}_j : \mathcal{R}_i \neq \mathcal{R}_j \wedge ((\mathcal{D}_i^w \cap \mathcal{D}_j \neq \emptyset) \vee (\mathcal{D}_i \cap \mathcal{D}_j^w \neq \emptyset)) : x_{i,c} + x_{j,c} \leq 1$

As in Sec. 4.2, the number of colors in the ILP is given by the number of vertices in the graph.

Example 5 (continued) This updated Constraint 2 results in the following constraints for this set of requests:

$$x_{1,1} + x_{4,1} \leq 1$$
$$x_{2,1} + x_{3,1} \leq 1$$
$$x_{2,1} + x_{4,1} \leq 1$$
$$x_{3,1} + x_{4,1} \leq 1$$
repeat above four inequalities for Colors 2-4

6 Hierarchical Organization

Our initial approach to determining concurrency groups resulted in each request being satisfied with the same frequency. Here we explore adding a layer of hierarchy to the request-management scheme to alter the frequency with which requests are satisfied. We begin by considering a group of six requests: the five requests from Ex. 4 and one additional request.

Example 7 Consider the task set with the requests from Ex. 4 and a sixth request, $\mathcal{R}_6$, for $\mathcal{D}_6 = \{\ell_a, \ell_e\}$ with a critical-section length of at most $L_6 = 55$ time units. Using the approach described in Secs. 3.1 and 4.2, we determine that four concurrency groups as shown in Fig. 10 are required (adding $\mathcal{G}_4 = \{\mathcal{R}_6\}$ to the three groups used in Ex. 4). 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. 4, each request in this set may experience 155 time units of blocking.

In this section, we propose a modification to the satisfaction order of concurrency groups that can lower the worst-case blocking for *most* requests at the cost of increasing the worst-case blocking for a *few* requests.

6.1 Hierarchical Request Satisfaction

Under this scheme, a set of *slots*[2] becomes active in a round-robin-like fashion like the group arbitration process described in Sec. 3.3. When a given slot becomes active, one of the groups assigned to that slot is granted an active phase. Groups within a slot compete as in Sec. 3.3, but now a group competes with only the other groups in the same slot. Each group belongs to exactly one slot. We denote the set of all groups belonging to Slot a as $\mathcal{S}_a$.

[2] Note that this work presents a change in the definition of slot from that in [43].

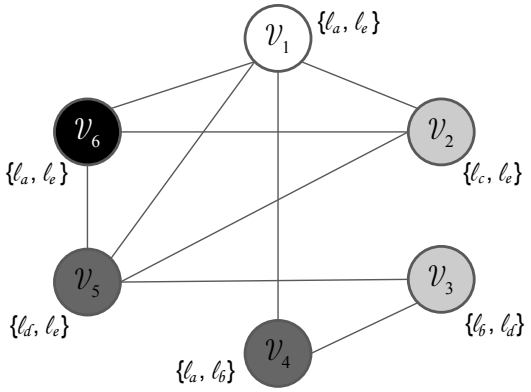


Fig. 10 Four concurrency groups for requests $\mathcal{R}_1$ to $\mathcal{R}_6$: $\mathcal{G}_1 = \{\mathcal{R}_1\}$, $\mathcal{G}_2 = \{\mathcal{R}_2, \mathcal{R}_3\}$, $\mathcal{G}_3 = \{\mathcal{R}_4, \mathcal{R}_5\}$ and $\mathcal{G}_4 = \{\mathcal{R}_6\}$.

Example 7 (continued) The concurrency groups depicted in Fig. 10 may be assigned to slots such that $\mathcal{S}_1 = \{\mathcal{G}_1\}$, $\mathcal{S}_2 = \{\mathcal{G}_2, \mathcal{G}_4\}$, and $\mathcal{S}_3 = \{\mathcal{G}_3\}$.

All groups in $\mathcal{S}_a$ may compete to *occupy* Slot a in its active phase. We add to the rules stated in Sec. 3.3 to capture this new structure. All of the rules in Sec. 3.3 apply without modification (*e.g.*, note that G8 refers to any waiting group belonging to any slot) except for G3, which we replace with a modified version below.

We begin by stating the rules governing the coordination between the slots.

G9 A slot is either *active*, *waiting*, or *inactive*, and at most one slot is active at any time.

G10 A slot is inactive if all of its groups are inactive.

G11 A waiting slot becomes active once all slots that were active or waiting when this slot entered the waiting state have completed a single phase of execution.

The following rules govern how the groups interact with their respective slots.

G12 A group may only be active if it occupies its slot.

G13 At most one group can occupy a slot at a time.

G14 If a group belonging to an inactive slot becomes active, then the group immediately occupies the slot, and the slot becomes active if no slot is active, or waiting if there is an active slot.

G15 When a slot becomes active, the group that occupies that slot becomes active.

G16 When the group that occupies the active slot completes, the active phase of the slot completes; if there are waiting slots, this slot enters the waiting state if there are waiting groups in this slot, or enters the inactive state otherwise.

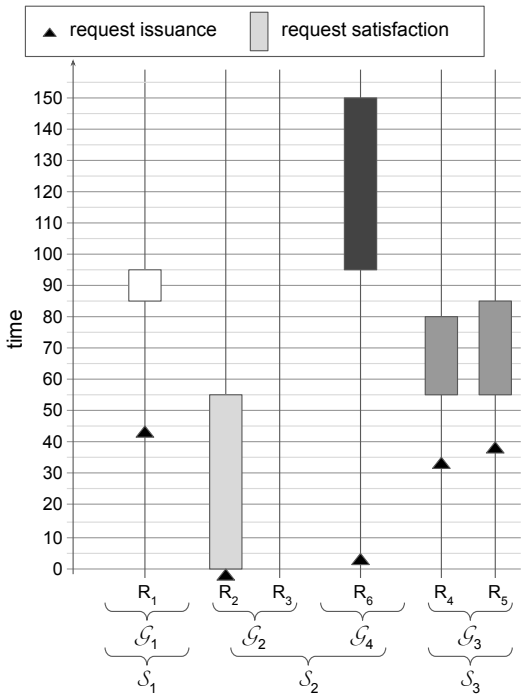


Fig. 11 An illustration of execution under the hierarchical approach.

We illustrate these rules with an example depicted in Fig. 11 and described below.

Example 7 (continued) Before time $t = 0$, there are no active requests. Thus all groups are inactive and all slots are inactive. At $t = 0$, $\mathcal{R}_2$ is issued, and by Rules G2, G12, G14, and G15, $\mathcal{G}_2$ occupies Slot 2, Slot 2 becomes active, and $\mathcal{G}_2$ becomes active. By Rule G4, $\mathcal{R}_2$ is therefore satisfied immediately.

The new version of Rule G3 is:

G3' A waiting group *occupies its slot* once all groups that were active or waiting *in its slot* when this group entered the waiting state have completed a single phase of execution.

Example 7 (continued) As shown in Fig. 11, at $t = 5$, $\mathcal{R}_6$ is issued. Because $\mathcal{G}_2$ occupies Slot 2, $\mathcal{G}_4$ can occupy the slot only after $\mathcal{G}_2$ has completed an active phase (Rule G3'), which occurs at $t = 55$. At this time, by Rule G16, Slot 2 enters the waiting state, so by Rule G11, Slot 2 cannot become active until both Slot 3 and Slot 1 have completed a single phase, which occurs at $t = 95$.

As described in the rules above, we enforce a FIFO ordering among groups competing for a given slot; the group with the earliest-issued active request occupies the slot until all requests that were active when the group became

active have completed. This introduces an additional layer of hierarchy and additional blocking for these requests; a request must now wait until its group occupies its slot and then for its slot to become active before it can be satisfied.

6.2 Bounding Blocking

To reason about the worst-case acquisition delay under this hierarchical approach, we define $L_{max}^{\mathcal{S}_a}$ such that it upper-bounds the length of one phase of Slot a; we let $L_{max}^{\mathcal{S}_a} = \max_{\mathcal{G}_g \in \mathcal{S}_a} L_{max}^{\mathcal{G}_g}$.

Lemma 3 *If there is an active request in a group in $\mathcal{S}_a$, Slot a will become active within $\sum_{b \neq a} L_{max}^{\mathcal{S}_b}$ time units.*

Proof Slot a must be either active or waiting, as at least one of its groups is not inactive (Rules G9 and G10). If Slot a is waiting, it will become active once all slots that were active or waiting when this slot entered the waiting state have completed a single phase of execution (Rule G11). The bound of $\sum_{b \neq a} L_{max}^{\mathcal{S}_b}$ follows directly from the definition of $L_{max}^{\mathcal{S}_a}$. □

Example 7 (continued) At $t = 45$, $\mathcal{R}_1$ is issued, Slot 2 is active, and Slot 3 is waiting. By Rule G14, Slot 1 enters the waiting state at $t = 45$. By Rule G11, Slot 1 will become active once Slot 2 and Slot 3 each complete a phase of execution. By Lemma 3, this will occur within $60 + 30 = 90$ time units.

Theorem 4 *The worst-case acquisition delay a request $\mathcal{R}_i$ in group $\mathcal{G}_g$ in $\mathcal{S}_a$ may experience is upper-bounded by $|\mathcal{S}_a| \cdot \sum_b L_{max}^{\mathcal{S}_b}$.*

Proof When $\mathcal{R}_i$ is issued, if $\mathcal{G}_g$ does not occupy its slot, then a different group, $\mathcal{G}_d$, occupies the slot. $\mathcal{G}_d$ becomes active within $\sum_{b \neq a} L_{max}^{\mathcal{S}_b}$ time units (Lemma 3 and Rule G15), and then is active for up to $L_{max}^{\mathcal{G}_d} \leq L_{max}^{\mathcal{S}_a}$ time units. By Rule G3', $\mathcal{G}_g$ occupies its slot once all groups that were active or waiting in its slot when it entered the waiting state have completed a phase of execution. There are at most $|\mathcal{S}_a| - 1$ such groups, and as reasoned above with $\mathcal{G}_d$, each takes at most $\sum_{b \neq a} L_{max}^{\mathcal{S}_b} + L_{max}^{\mathcal{S}_a} = \sum_b L_{max}^{\mathcal{S}_b}$ time units to complete a phase of execution. Once $\mathcal{G}_g$ occupies its slot, it becomes active within $\sum_{b \neq a} L_{max}^{\mathcal{S}_b}$ time units (Lemma 3 and Rule G15). Thus $\mathcal{R}_i$ experiences an acquisition delay of up to $(|\mathcal{S}_a| - 1) \cdot (\sum_b L_{max}^{\mathcal{S}_b}) + \sum_{b \neq a} L_{max}^{\mathcal{S}_b} \leq |\mathcal{S}_a| \cdot \sum_b L_{max}^{\mathcal{S}_b}$.

Otherwise, $\mathcal{G}_g$ does occupy its slot when $\mathcal{R}_i$ is issued. Then $\mathcal{G}_g$ is either waiting or active (Rule G2). If $\mathcal{G}_g$ is waiting, it becomes active within $\sum_{b \neq a} L_{max}^{\mathcal{S}_b}$ time units (Lemma 3 and Rule G15), at which time $\mathcal{R}_i$ is satisfied (Rule G4).

If instead $\mathcal{G}_g$ is active, then $\mathcal{R}_i$ is either satisfied immediately (resulting in no acquisition delay) or there must be waiting groups (Rule G8). If there are waiting groups, by Rule G8, $\mathcal{R}_i$ will not be satisfied until the next active phase of $\mathcal{G}_g$. The current active phase of $\mathcal{G}_g$ finishes within $L_{max}^{\mathcal{G}_g} \leq L_{max}^{\mathcal{S}_a}$ time units before entering the waiting state. The remaining delay $\mathcal{R}_i$ incurs depends on whether there are other groups in $\mathcal{S}_a$ that are waiting.

If there are not other waiting groups in its slot, $\mathcal{G}_g$ occupies its slot and $\mathcal{R}_i$ is satisfied when $\mathcal{G}_g$ becomes active, within $\sum_{b \neq a} L_{max}^{\mathcal{S}_b}$ time units (Lemma 3 and Rules G4 and G15). Thus $\mathcal{R}_i$'s acquisition delay would be bounded by $L_{max}^{\mathcal{S}_a} + \sum_{b \neq a} L_{max}^{\mathcal{S}_b} = \sum_b L_{max}^{\mathcal{S}_b}$ time units.

Finally, if there are other waiting groups belonging to $\mathcal{S}_a$, then once $\mathcal{G}_g$ finishes its active phase, another group occupies its slot (Rule G3'). Then, as above, at most $|\mathcal{S}_a| - 1$ groups occupy $\mathcal{S}_a$ before $\mathcal{G}_g$ again occupies it; these complete within $(|\mathcal{S}_a|-1)\cdot(\sum_b L_{max}^{\mathcal{S}_b})$ time units. Then $\mathcal{G}_g$ becomes active again within $\sum_{b \neq a} L_{max}^{\mathcal{S}_b}$ time units (by Lemma 3 and Rule G15). When $\mathcal{G}_g$ becomes active, $\mathcal{R}_i$ is satisfied (Rule G4). Thus, the worst-case acquisition delay of $\mathcal{R}_i$ is bounded by $L_{max}^{\mathcal{S}_a} + (|\mathcal{S}_a|-1)\cdot(\sum_b L_{max}^{\mathcal{S}_b}) + \sum_{b \neq a} L_{max}^{\mathcal{S}_b} = |\mathcal{S}_a| \cdot \sum_b L_{max}^{\mathcal{S}_b}$. □

Example 7 (continued) As depicted in Fig. 11, when $\mathcal{R}_6$ is released at $t = 5$, $\mathcal{G}_4$ does not occupy its slot. With this new hierarchical approach, instead of being satisfied after all active groups have completed a phase of execution (here, only $\mathcal{G}_2$), it is satisfied at $t = 95$. This acquisition delay is captured in Theorem 4: $\mathcal{R}_6$ has acquisition delay bounded by $|\{\mathcal{G}_2, \mathcal{G}_4\}| \cdot \sum_{b=1}^{3} L_{max}^{\mathcal{S}_b} = 2 \cdot (10 + 60 + 30) = 200$ time units (as do $\mathcal{R}_2$ and $\mathcal{R}_3$). This benefits $\mathcal{R}_1$, $\mathcal{R}_4$, and $\mathcal{R}_5$, which now have acquisition delay bounded by $1 \cdot (10 + 60 + 30) = 100$ time units.

In essence, we can increase blocking for some requests in order to lower blocking for other requests. The decision of which groups of requests to map to the same slot can depend on multiple factors. In general, some tasks may be able to incur a higher amount of blocking and still meet their deadlines; this will depend on specific details of each task.

6.3 Assigning Groups to Slots

Although Theorem 4 upper-bounds the acquisition delay each request may experience, it does not guide how to assign groups to slots. Here we provide some intuition behind assignment decisions and a few possible approaches.

In general, the benefit of adding a layer of hierarchy in Ex. 7 is that less blocking is incurred in the system as a whole: three requests may incur up to 200 time units of blocking and three up to 100 time units as opposed to all six incurring up to 155 time units each. Thus, a first approach would be to form an optimization process that minimizes the summed blocking. However, lowering total blocking across all requests ignores the periods of the tasks issuing these requests and each task's capacity to incur higher blocking and still meet its deadlines. Thus, an approach that minimizes the summed blocking may ignore crucial features for schedulability.

A second approach would be to pre-select tasks to belong to groups that share a slot with at least one other group (with the remaining tasks being assigned to groups that would not share a slot). The underlying motivation is that the slot-sharing groups should be comprised of tasks that are able to incur additional blocking. Without knowing the resulting blocking ahead of time,

these tasks could be selected by their low utilization or high period, both of which are properties that give additional flexibility for incurring higher blocking. Once the tasks are separated, the groups for each set could be determined with the ILP in Sec. 5. Thus two distinct ILPs would be solved to yield two sets of groups. The groups could then be assigned to slots randomly, ensuring that the groups from the first set of tasks always share a slot and those from the second never do.

The final approach that we present enforces that each slot has either one or two groups and maximizes the minimum anticipated relative slack for tasks of each group. Without the considerations of blocking, the *slack* of a task τ_i is $T_i - C_i$ (for implicit-deadline tasks); this captures the capacity τ_i has to incur delays and still meet its deadline. The anticipated relative slack incorporates the expected blocking. The intuition behind maximizing the minimum anticipated relative slack is to maximize the buffer tasks have to incur delay and still meet their deadlines. We specify this as an optimization problem and describe the approach in more detail as we present each constraint.

As before, we must encode a maximum number of colors. We choose the number of request-issuing tasks, denoted *num_req*. We also pre-select which groups will share a slot: all groups with a number at most $split = \lfloor \frac{num\_req}{4} \rfloor \cdot 2$ will share a slot. Group $\mathcal{G}_g$ will be in $\mathcal{S}_{\lfloor \frac{g}{2} \rfloor + (g \mod 2)}$ if $g \leq split$ or $\mathcal{S}_{g - \frac{split}{2}}$ otherwise. We leave the exploration of alternative choices of *split* to future work.

We now describe the constraints we apply. We reuse the constraints from Sec. 3.2 to enforce the basic coloring restrictions.

Constraint 1 $\forall i : \sum_c x_{i,c} = 1$

Constraint 2 $\forall c \; \forall \mathcal{R}_i, \mathcal{R}_j : \mathcal{R}_i \neq \mathcal{R}_j \wedge \mathcal{D}_i \cap \mathcal{D}_j \neq \emptyset : x_{i,c} + x_{j,c} \leq 1$

Next we constrain a set of variables to represent the maximum duration of each slot: $duration_a$ represents $L_{max}^{\mathcal{S}_a}$. Based on our construction of a hierarchical approach, each slot has either one or two groups that impact $L_{max}^{\mathcal{S}_a}$. This results in the following three constraints.

Constraint 3 $\forall i \; \forall a : a \leq \frac{split}{2} : duration_a \geq x_{i,2a-1} \cdot L_i$

Constraint 4 $\forall i \; \forall a : a \leq \frac{split}{2} : duration_a \geq x_{i,2a} \cdot L_i$

Constraint 5 $\forall i \; \forall a : a > \frac{split}{2} : duration_a \geq x_{i,split+a} \cdot L_i$

Next we constrain the variable *summed_duration*, which upper-bounds the sum in Theorem 4.

Constraint 6 $summed\_duration \geq \sum_b duration_b$

Our final constraints are those on the anticipated relative slack for each $\mathcal{G}_g$, denoted $slack_g$. Note that the groups that share a slot ($g \leq split$) have a factor of 2 multiplying *summed_duration*, while the other groups do not. This reflects the upper bound of blocking presented in Theorem 4.

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.

Constraint 7 $\forall i\ \forall c \leq split : slack_c \leq \frac{T_i - (C_i + 2 \cdot summed\_duration) x_{i,c}}{T_i}$

Constraint 8 $\forall i\ \forall c > split : slack_c \leq \frac{T_i - (C_i + \cdot summed\_duration) x_{i,c}}{T_i}$

We then seek to maximize the summed slack over all groups. The anticipated per-group relative slack can be between zero and one; a color with no tasks assigned to it has a minimum per-group relative slack of one.

Objective $max \sum_c slack_c$

The above optimization can be transformed in a straightforward manner with the approach shown in Sec. 5 to handle mixed requests.

7 Evaluation

Our evaluation of the CGLP is comprised of two parts: measuring the time required for the offline group formation and comparing its online performance to prior real-time locking protocols in a schedulability study. In our experiments, we explored a broad space of task-system parameters, varying the individual task utilization, the period, the percentage of tasks that issue requests, the critical-section lengths, the probability that a given request is nested, the number of resources requested for a nested request, and the probability that a nested request is mixed; named value sets are listed in Table 2, and the set of parameters used for our schedulability study are in Table 3. If a nested request is mixed, it is randomly assigned to require read access to half of its resources and write access to its other resources. We define a *scenario* to be a setting of each of the above parameters.

7.1 Analysis of Offline Component

Here, we evaluate three possibles approaches to implementing the offline component of the CGLP. In Sec. 3.2, we presented an ILP to assign concurrency

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
Mixed Probability	0, 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.

Task Utilization	Average Number of Requests	Average k
Heavy	23.4	3.7
Medium	64.7	6.7
Medium-Light	291.6	19.8

Table 4 Average size of a graph coloring problem for a system with total utilization of 16.

groups such that the number of groups is minimized. We denote this approach BasicILP. In Sec. 4.2, we presented an optimization problem to instead minimize blocking, and we denote this DurationILP. Finally, in Sec. 5.2, we showed how the above two optimization problems can be modified to account for mixed requests in a more fine-grained manner. Here, we explore this modification to the duration-based ILP, which we denote RW-DurationILP.

In this section, we examine how long each of these offline components takes to determine the concurrency groups. We compare each of these without considering the additional impacts of the possible hierarchical approaches. We leave evaluation of those to future work.

To give a sense of the scale of these problems, we generated random task sets with total utilization of 16, in which every task issued a request and nested requests required four random resources.

For each per-task utilization, we varied the nested probability and report the average number of requests and average minimum number of colors (using BasicILP) across 50 task sets in Table 4. Note that for task sets with heavy per-task utilization, fewer tasks are needed to reach the given target system utilization; conversely, with medium-light per-task utilization, many more tasks are needed to reach the target system utilization, and thus more colors are needed.

As described in our specification of each ILP, we must create variables for each color that might need to be used. In order to improve performance of BasicILP, we restrict this number (thereby reducing the number of variables, and thus, the problem size) based on the result of a greedy coloring algorithm. The greedy coloring approach obtains a *proper* k-coloring of a graph. Although the value of k obtained from greedy coloring is not necessarily minimal, it

provides a better upper bound on the chromatic number than simply using the total number of requests. The greedy approach we applied begins with a number of colors equal to the number of vertices in the graph, and each color is labeled with a numerical identifier. Each vertex is assigned the lowest numbered color that does not appear among its already colored neighbors.

To test how long it takes to determine concurrency groups for a given task set, we generated random task sets across the space of all scenarios. For each task set, we timed the setup time and the time required to solve each ILP with an ILP solver [33]. In Fig. 12, we show the 95^{th} percentile of the solve time for each ILP in a scenario with heavy per-task utilization, short periods, and 100% of tasks issuing requests with long critical-section lengths. Each request was nested with the probabilities shown. Nested requests required access to four randomly selected resources, and the probability of a request being nested was 0.8. For each scenario, we generated 100 task sets.

Obs. 1 *Although 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.*

This is demonstrated in Fig. 12, which depicts a scenario with higher-than-average ILP solve times for tasks with heavy utilization; in this scenario, the maximum 95th percentile solve time was still less than 3s. Based on these results, we set a timeout on the ILP solver for each ILP when running our schedulability study (described in more detail in Sec. 7.2). For task sets with heavy per-task utilization, we used 0.1s for BasicILP and 2.7s for the two duration-based ILPs. In our schedulability experiments, these thresholds were never exceeded. For task sets with medium per-task utilization, we enforced a timeout of 0.1s for BasicILP and 120s for the duration-based ILPs. Across 16.7 million task sets, one of these limits was exceeded only 153 times.

Obs. 2 *The time to solve each ILP depends on the number of requests and the connectivity of the graph formed from those requests.*

This observation is supported by looking at a range of factors. The per-task utilization determines the number of tasks in the systems, and the percentage of tasks which issue requests determines the total number of requests. Systems with higher total utilization are comprised of more tasks (and thus more requests). The connectedness of the graph depends on the probability of a given request being nested and the number of resources required by nested requests. We illustrate this with a small set of scenarios: in Fig. 12, we show the time required to solve each ILP for a scenario in which each nested request requires four resources. As shown in Fig. 12, the nested probability has a significant impact on the time required to solve each ILP.

Additionally, we note that the solution times for the duration-based ILPs are significantly higher than those for BasicILP. We hypothesize that the solution times for DurationILP and RW-DurationILP are higher partially due to having so many additional variables; we cannot use the greedy coloring to

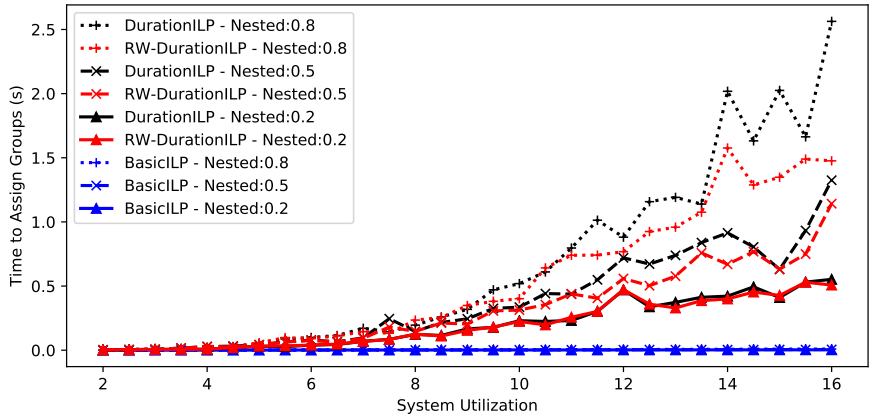


Fig. 12 Average time to solve each ILP. Each data point represents the 95^{th} percentile from 100 random task sets.

determine the minimum number of colors required to enable the ILP to minimize blocking. (Recall the discussion in Sec. 4.1 that minimizing colors may not minimize duration.) Therefore, we use the number of requests as a safe upper bound on the number of colors.

7.2 Schedulability Study

We evaluated the CGLP as a whole on the basis of schedulability with a large-scale schedulability study. We compared the 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 General C-RNLP (G-C-RNLP) grants resource access in a contention-sensitive manner on a per-request basis, and the Uniform C-RNLP (U-C-RNLP) does so by granting access to sets of requests (these sets are determined dynamically based on issuance order). We also compare the CGLP to the RNLP and to a simple group lock.[3]

Experimental setup. We conducted schedulability experiments using SchedCAT [2], 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 [11]. We inflated the execution time of tasks based on the locking protocol overhead and blocking their requests may incur, as described in [14]. In this

[3] We use a single MCS lock [38] to protect all resources for the group lock. We leave it to future work to compare to a set of resource-ordered locks.

Protocol	Worst-Case Acquisition Delay	Total Overhead (μs)
CGLP	$\sum_c L_{max}^{\mathcal{G}_c}$	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	$(m - 1) \cdot L_{max}$	13.5
MCS	$(m - 1) \cdot L_{max}$	0.7

Table 5 Blocking bounds and overhead of each protocol. For the C-RNLP bounds, 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.)

context, overhead refers to the time a request spends adding or removing itself from the lock queue(s), and blocking is the time spent waiting for the request to be satisfied once it has been enqueued.

The blocking bounds and overhead of each protocol are summarized in Table 5. Stated overhead values are the 99^{th} percentile of measurements taken on a dual-socket, 8-cores-per-socket machine with 32 GB of DRAM, running Ubuntu 16.04. To maximize observed overhead, each task submitted a new request immediately after completing the prior one. Each request was for four of 64 total resources, and there were up to sixteen requests active at once.

As discussed above, the offline portion of the CGLP can be implemented with several different approaches; we show the results for three of them here. We set a timeout value for the ILP solver (see Sec. 7.1) for each approach. If this limit is ever exceeded, we instead assign groups with a greedy coloring approach.

When calculating the blocking bounds for the CGLP in the schedulability study, we use the expression presented in Theorem 3, in combination with additional refinements to bound the acquisition delay $\mathcal{R}_i$ can experience.

For example, because we focus on a spin-based system with m processors, at most $m-1$ other requests may be active at the time of $\mathcal{R}_i$'s issuance. Thus, if there are more than $m-1$ concurrency groups, only the $m-1$ largest $L_{max}^{\mathcal{G}_c}$ values should be counted toward the blocking $\mathcal{R}_i$ may experience. Additionally, $\mathcal{R}_i$ may have the highest critical-section length of its group. In this case, when calculating the maximum blocking it may experience due to an active phase of its group, the second-highest critical-section length of a request in its group should be used.

We applied similar refinements to the C-RNLP variants. Again, at most $m-1$ other requests may be active for each variant. Tighter analysis can also be applied to determine the longest critical sections that may block $\mathcal{R}_i$; this is different between the two variants, but for both we incorporated methods that limit the number of times to include the largest L_j terms toward the blocking of $\mathcal{R}_i$. These methods rely on how many times a request $\mathcal{R}_j$ could be issued while $\mathcal{R}_i$ is active (based on the period of each task) and the structure of the given protocol, especially with regard to a requests that share a set of resources with $\mathcal{R}_i$.

Our schedulability study considered the 1,152 scenarios listed in Table 3; common trends are discussed here, and the full set of plots is available online along with the code [44]. For each scenario, 1000 task systems were generated for every value of system utilization; we plot the percentage of these that are schedulable when no synchronization delay is accounted for (NOLOCK) and when synchronization delay results from one of the protocols we compare.

To compare the schedulability results under different protocols, we computed the *schedulable utilization area* (SUA) of each by approximating the area under the curve with a midpoint Riemann sum. We used 0.05 as a threshold for determining if two protocols performed about as well as each other; if their SUAs in a given scenario were less than 0.05 different, we report that they performed roughly the same under that scenario.

Comparison of the CGLP to existing protocols. We now present our general observations along with graphs that showcase key trends from this study. We begin by considering the 288 scenarios in which the probability of a nested request being mixed is zero and comparing the CGLP approaches to existing protocols; the analysis of existing protocols does not handle mixed requests and instead must treat such requests as write requests.

We state key trends and illustrate these with representative graphs. In Fig. 13 we show the results of four scenarios. Each point depicts the fraction of task sets of a given system utilization that were deemed schedulable.

Obs. 3 *The CGLP approaches perform as well or better in most scenarios.*

This is illustrated in Fig. 13(a-c). For task systems with moderate critical-section lengths, the CGLP approaches always have the highest SUA (tied or alone). For medium per-task utilization, the CGLP approaches perform as well or better than other approaches in 70.1% of scenarios, and in 54.9% of scenarios, the CGLP outperforms any existing approach.

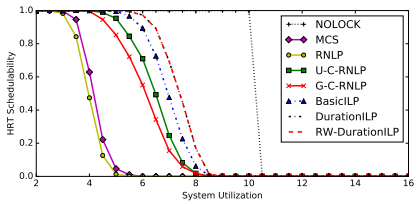
Obs. 4 *When a CGLP approach is not the best, the G-C-RNLP is the protocol that results in the highest SUA.*

In 33.7% of scenarios the G-C-RNLP outperforms all other protocols. In 97.9% of these scenarios, periods are short. One such scenario is illustrated in Fig. 13d.

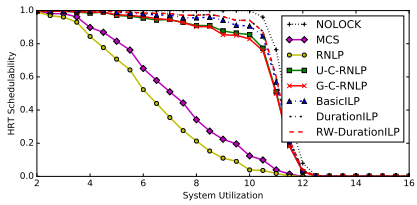
Comparison between CGLP variants. We now compare the different CGLP approaches. To do so, we consider all values of mixed probability used in our schedulability study. We highlight the results of two scenarios in Fig. 14, both of which have a nested probability of 0.5, a nesting depth of 4, a mixed probability of 0.8, and 100% of the tasks issued requests.

Obs. 5 *DurationILP is never worse than BasicILP; frequently it is better.*

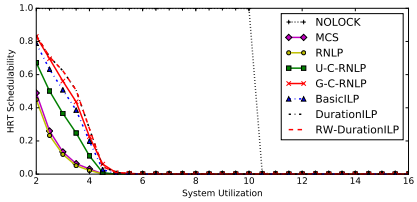
DurationILP is better in 52.5% of scenarios. In each scenario in Figs. 13 and 14, DurationILP is better than BasicILP, as determined by comparing SUAs.



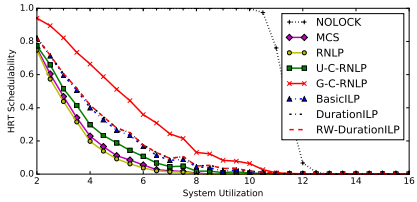
(a) Medium utilization, moderate critical-section length



(b) Heavy utilization, moderate critical-section length

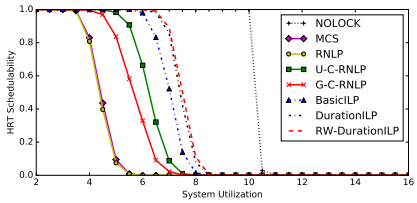


(c) Medium utilization, weighted bimodal critical-section length

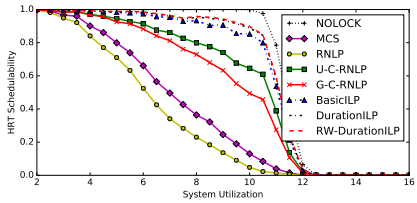


(d) Heavy utilization, weighted bimodal critical-section length

Fig. 13 Scenarios in which periods were short, nested probability was 0.2, nesting depth was 4, mixed probability was 0, and 100% of tasks issued requests.



(a) Medium per-task utilization, long periods, long critical-section lengths



(b) Heavy per-task utilization, short periods, moderate critical-section lengths

Fig. 14 For this scenario, nested probability was 0.5, nesting depth was 4, mixed probability was 0.8, and 100% of tasks issued requests.

Obs. 6 *While RW-DurationILP frequently has a slightly higher SUA than DurationILP, it rarely is significantly better.*

The SUA of RW-DurationILP exceeds that of DurationILP by the 0.05 threshold in only eight scenarios, all with medium per-task utilization, nested probability of 0.5, and mixed probability of 0.8. One such scenario is shown in Fig. 14a. A scenario in which the SUA of RW-DurationILP exceeds that of DurationILP by less than 0.05 is shown in Fig. 14b.

8 Related Work

There is a large body of work aimed at developing locking protocols for multiprocessor systems [3–7,12–15,17–22,24,25,27–32,35,36,39–41,46–60]. However, most of these approaches are limited to systems without lock nesting. Here, we focus on those protocols that do allow nesting. (Lock nesting can be trivially supported by redefining resources such that no nesting occurs; using such a coarse-grained group lock can cause significant unnecessary blocking between tasks that do not actually share any resources. We compare to one such approach, the MCS lock [38], but focus instead on describing fine-grained locking protocols that already support lock nesting.)

One synchronization approach that allows nested access to resources is the *multiprocessor bandwidth inheritance protocol* (M-BWI) [28,29]. Another approach is *MrsP* [22,32,60]. Both the M-BWI and MrsP require an ordering on nested resource acquisition to prevent deadlock.[4] A straightforward bound on the blocking a request may experience when deadlock is prevented by resource ordering is exponential in the number of resources [49]. Computing a tight bound on worst-case blocking is NP-hard when nesting is allowed [58].

The *real-time nested locking protocol (RNLP)* [35,41,42,52–55] family of protocols also supports nested requests. Of the protocols mentioned, most do not handle the Transitive Blocking Chain Problem. Those that do are the fast RW-RNLP [42] and the C-RNLP [35]. 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 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 blocking chain 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 $\mathcal{D}_i$ 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.

Other related protocols are reader-reader protocols; the CGLP builds on the notion of a reader-reader locking protocol (a synchronization mechanism that manages resource access between groups of read requests [41]). The CGLP allows one group of requests access at a time; any requests from another group

[4] This ordering refers to the order in which resources must be acquired by a given task, not the order in which requests are satisfied [26,34].

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^3LP [42], which coordinates three groups of read requests. Existing work [41, 42] 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 hinges on considering 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 [9, 10, 23, 37].

In this work, we focus primarily on nested write requests and briefly describe how the CGLP approaches can be modified to allow fine-grained access for mixed requests. Existing work allows a nested write locking protocol to be combined with other protocols in a modular fashion to produce a reader/writer locking protocol [40, 42]. Other work has looked at the reader/writer sharing paradigm [19, 20, 55] or other sharing paradigms like k-exclusion [20, 27, 56], among others [51].

9 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 presented multiple approaches to this offline component and proved upper bounds on acquisition delay.

The offline approaches to forming concurrency groups optimize for one of a few conditions: minimum number of groups, minimum per-request blocking by accounting for critical-section lengths, or maximum relative slack. These allow worst-case blocking to be tuned based on task parameters.

We showed that, for many task systems, the offline formation of concurrency groups can be achieved by an ILP solver very quickly. We also evaluated the CGLP on the basis of schedulability and showed that in many scenarios the CGLP outperforms existing protocols.

As future work, this protocol will be incorporated as a component of a larger protocol to merge work that handles non-nested requests and read requests efficiently with the efficient management of nested and mixed-type requests provided by the CGLP.

References

1. AUTOSAR Release 4.4, Classic Platform, Specification of Operating System. `https://www.autosar.org/` (2019). Accessed: 2019-05-20
2. SchedCAT: Schedulability test collection and toolkit. `https://github.com/brandenburg/schedcat` (2019). Accessed: 2019-02-07
3. Afshar, S., Behnam, M., Bril, R., Nolte, T.: Flexible spin-lock model for resource sharing in multiprocessor real-time systems. In: SIES '14 (2014)
4. Afshar, S., Behnam, M., Bril, R., Nolte, T.: An optimal spin-lock priority assignment algorithm for real-time multi-core systems. In: RTCSA '17 (2017)
5. Afshar, S., Behnam, M., Bril, R., Nolte, T.: Per processor spin-based protocols for multiprocessor real-time systems. Leibniz Transactions on Embedded Systems **4**(2) (2018)
6. Afshar, S., Behnam, M., Nolte, T.: Integrating independently developed real-time applications on a shared multi-core architecture. ACM SIGBED Review '13 (2013)
7. Afshar, S., Khalilzad, N., Nemati, F., Nolte, T.: Resource sharing among prioritized real-time applications on multiprocessors. ACM SIGBED Review '15 (2015)
8. Bacon, D., Konuru, R., Murthy, C., Serrano, M.: Thin locks: Featherweight synchronization for Java. In: PLDI '98 (1998)
9. Bandh, T., Carle, G., Sanneck, H.: Graph coloring based physical-cell-id assignment for lte networks. In: IWCMC '09 (2009)
10. Barnier, N., Brisset, P.: Graph coloring for air traffic flow management. Annals of operations research **130**(1-4) (2004)
11. Baruah, S.: Techniques for multiprocessor global schedulability analysis. In: RTSS '07 (2007)
12. Biondi, A., Brandenburg, B., Wieder, A.: A blocking bound for nested FIFO spin locks. In: RTSS '16 (2016)
13. Block, A., Leontyev, H., Brandenburg, B., Anderson, J.: A flexible real-time locking protocol for multiprocessors. In: RTCSA '07 (2007)
14. Brandenburg, B.: Scheduling and locking in multiprocessor real-time operating systems. Ph.D. thesis, University of North Carolina, Chapel Hill, NC (2011)
15. Brandenburg, B.: The FMLP+: An asymptotically optimal real-time locking protocol for suspension-aware analysis. In: ECRTS '14 (2014)
16. Brandenburg, B., Anderson, J.: Feather-trace: A lightweight event tracing toolkit. In: OSPERT '07 (2007)
17. Brandenburg, B., Anderson, J.: A comparison of the M-PCP, D-PCP, and FMLP on LITMUS$^{\mathrm{RT}}$. In: OPODIS '08 (2008)
18. Brandenburg, B., Anderson, J.: An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS$^{\mathrm{RT}}$. In: RTCSA '08 (2008)
19. Brandenburg, B., Anderson, J.: Spin-based reader-writer synchronization for multiprocessor real-time systems. Real-Time Systems **46**(1) (2010)
20. Brandenburg, B., Anderson, J.: Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In: EMSOFT '11 (2011)
21. Brandenburg, B., Anderson, J.: The OMLP family of optimal multiprocessor real-time locking protocols. Design Automation for Embedded Systems **17**(2), 277–342 (2013)
22. Burns, A., Wellings, A.: A schedulability compatible multiprocessor resource sharing protocol - MrsP. In: ECRTS '13 (2013)
23. Chaitin, G., Auslander, M., Chandra, A., Cocke, J., Hopkins, M., Markstein, P.: Register allocation via coloring. Computer languages **6**(1) (1981)
24. Chang, Y., Davis, R., Wellings, A.: Reducing queue lock pessimism in multiprocessor schedulability analysis. In: RTNS '10 (2010)
25. Chen, C., Tripathi, S.: Multiprocessor priority ceiling based protocols. Dept. of Computer Science, Univ. of Maryland. Tech. rep., CS-TR-3252, April (1994)
26. Dijkstra, E.: Two starvation free solutions to a general exclusion problem. EWD 625, Plataanstraat 5, 5671 Al Nuenen, The Netherlands (1977)
27. Elliott, G., Anderson, J.: An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems. Real-Time Systems **49**(2), 140–170 (2013)

28. Faggioli, D., Lipari, G., Cucinotta, T.: The multiprocessor bandwidth inheritance protocol. In: ECRTS '10 (2010)
29. Faggioli, D., Lipari, G., Cucinotta, T.: Analysis and implementation of the multiprocessor bandwidth inheritance protocol. Real-Time Systems **48**(6) (2012)
30. Gai, P., Di Natale, M., Lipari, G., Ferrari, A., Gabellini, C., Marceca, P.: A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform. In: RTAS '03 (2003)
31. Gai, P., Lipari, G., Di Natale, M.: Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In: RTSS '01 (2001)
32. Garrido, J., Zhao, S., Burns, A., Wellings, A.: Supporting nested resources in MrsP. In: Ada-Europe International Conference on Reliable Software Technologies '17 (2017)
33. Gurobi Optimization, L.: Gurobi optimizer reference manual (2018). URL `http://www.gurobi.com`
34. Havender, J.: Avoiding deadlock in multitasking systems. IBM systems journal **7**(2) (1968)
35. Jarrett, C., Ward, B., Anderson, J.: A contention-sensitive fine-grained locking protocol for multiprocessor real-time systems. In: RTNS '15 (2015)
36. Lakshmanan, K., Niz, D., Rajkumar, R.: Coordinated task scheduling, allocation and synchronization on multiprocessors. In: RTSS '09 (2009)
37. Marx, D.: Graph colouring problems and their applications in scheduling. Periodica Polytechnica Electrical Engineering **48**(1-2) (2004)
38. Mellor-Crummey, J., Scott, M.: Algorithms for scalable synchronization of shared-memory multiprocessors. Transactions on Computer Systems **9**(1) (1991)
39. Nemati, F., Behnam, M., Nolte, T.: Independently-developed real-time systems on multi-cores with shared resources. In: ECRTS '11 (2011)
40. Nemitz, C., Amert, T., Anderson, J.: Real-time multiprocessor locks with nesting: Optimizing the common case. In: RTNS '17 (2017)
41. Nemitz, C., Amert, T., Anderson, J.: Using lock servers to scale real-time locking protocols: Chasing ever-increasing core counts. In: ECRTS '18 (2018)
42. Nemitz, C., Amert, T., Anderson, J.: Real-time multiprocessor locks with nesting: Optimizing the common case. Real-Time Systems **55**(2) (2019)
43. Nemitz, C., Amert, T., Goyal, M., Anderson, J.: Concurrency groups: A new way to look at real-time multiprocessor lock nesting. In: RTNS '19 (2019)
44. Nemitz, C., Amert, T., Goyal, M., Anderson, J.: Concurrency groups: A new way to look at real-time multiprocessor lock nesting (extended version) (2020). URL `http://www.cs.unc.edu/~anderson/papers.html`
45. Palladino, S.: Modelling graph coloring with integer linear programming. `https://manas.tech/blog/2010/09/16/modelling-graph-coloring-with-integer-linear-programming.html` (2010)
46. Rajkumar, R.: Real-time synchronization protocols for shared memory multiprocessors. In: ICDCS '90 (1990)
47. Rajkumar, R.: Synchronization in real-time systems: A priority inheritance approach (1991)
48. Rajkumar, R., Sha, L., Lehoczky, J.: Real-time synchronization protocols for multiprocessors. In: RTSS '88 (1988)
49. Takada, H., Sakamura, K.: Real-time scalability of nested spin locks. In: RTCSA '95 (1995)
50. Wang, C., Takada, H., Sakamura, K.: Priority inheritance spin locks for multiprocessor real-time systems. In: ISPAN '96 (1996)
51. Ward, B.: Relaxing resource-sharing constraints for improved hardware management and schedulability. In: RTSS '15 (2015)
52. Ward, B.: Sharing non-processor resources in multiprocessor real-time systems. Ph.D. thesis, University of North Carolina, Chapel Hill, NC (2016)
53. Ward, B., Anderson, J.: Supporting nested locking in multiprocessor real-time systems. In: ECRTS '12 (2012)
54. Ward, B., Anderson, J.: Fine-grained multiprocessor real-time locking with improved blocking. In: RTNS '13 (2013)
55. Ward, B., Anderson, J.: Multi-resource real-time reader/writer locks for multiprocessors. In: IPDPS '14 (2014)

56. Ward, B., Elliott, G., Anderson, J.: Replica-request priority donation: A real-time progress mechanism for global locking protocols. In: RTCSA '12 (2012)
57. Wieder, A., Brandenburg, B.: On spin locks in AUTOSAR: Blocking analysis of FIFO, unordered, and priority-ordered spin locks. In: RTSS '13 (2013)
58. Wieder, A., Brandenburg, B.: On the complexity of worst-case blocking analysis of nested critical sections. In: RTSS '14 (2014)
59. Yang, M., Wieder, A., Brandenburg, B.: Global real-time semaphore protocols: A survey, unified analysis, and comparison. In: RTSS '15 (2015)
60. Zhao, S., Garrido, J., Burns, A., Wellings, A.: New schedulability analysis for MrsP. In: RTCSA '17 (2017)