EFFICIENT SYNCHRONIZATION FOR REAL-TIME SYSTEMS WITH NESTED RESOURCE ACCESS

Catherine E. Nemitz

A dissertation submitted to the faculty at the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2021

Approved by:

James H. Anderson

Andrea Bastoni

Björn B. Brandenburg

Samarjit Chakraborty

F. Donelson Smith

ABSTRACT

Catherine E. Nemitz: Efficient Synchronization for Real-Time Systems with Nested Resource Access
(Under the direction of James H. Anderson)

Real-time systems are comprised of tasks, each of which must be guaranteed to meet its timing requirements. These tasks may request access to shared system components, called *resources*. Each such request may experience delays before being granted resource access. These delays can be separated into two categories: **(i)** those caused by the order in which tasks are granted resource access, and **(ii)** those caused by the time it takes to coordinate this ordering. If these delays are too large, a task may be unable to meet its timing requirements.

Tasks can require access to multiple resources concurrently, acquiring these resources in a nested fashion. This *nested resource access* can cause significant delays to tasks; these delays can far exceed those when only a single resource is required at a time, as certain request orderings cause delays between tasks that do not share any resources.

This dissertation presents locking protocols and a protocol-independent approach to mitigate resource access delays. Nested resource access can increase delays for all requests, including non-nested requests. The first protocol eliminates these additional delays by separating requests by type and creating a fast-path mechanism for non-nested requests. The next two protocols both reduce delays by reordering requests. One protocol reorders requests as they are issued, and the other uses an offline process to determine which requests may execute concurrently. These three protocols were compared to prior approaches in an evaluation across a range of task systems; all three protocols resulted in more task systems guaranteed to meet their timing requirements. Finally, a protocol-independent approach reduces delays by using a designated task to execute the locking protocol on behalf of other tasks. When applied to two protocol variants, this approach significantly reduced delays.

ACKNOWLEDGEMENTS

I am so very grateful for the many people who made this dissertation possible and supported me along the way. The completion of this dissertation would not have been possible without all of you.

First, I want to thank my adviser, Jim Anderson, who encouraged me to consider pursuing a doctorate, gave me the opportunity to join the group, and guided me over the past six years. I would also like to thank my dissertation committee members, Andrea Bastoni, Björn Brandenburg, Samarjit Chakraborty, and Don Smith, for all of their feedback.

I want to express my gratitude to all of my coauthors and collaborators: Tanya Amert, Nathan Burrow, Shai Caspin, Pontus Ekberg, Manish Goyal, Claire Nord, Brittany Subialdea, Bryan Ward, Kecheng Yang, and Ming Yang, many of whom have been members of the real-time systems group at UNC. This dissertation would not have been possible without their help. I have also appreciated the discussions with and feedback from the other members of the real-time systems group: Shareef Ahmed, Joshua Bakita, Lee Barnett, Micaiah Chisholm, Calvin Deutschbein, Zhishan Guo, Clara Hobbs, Namhoon Kim, Vance Miller, Mac Mollison, Sims Osborne, Nathan Otterness, Sarah Rust, Abhishek Singh, Stephen Tang, Peter Tong, Sergey Voronov, and Tyler Yandrofski, as well as the interactions with members of the broader Cyber-Physical Systems Group. I want to especially recognize Tanya, who has been an amazing coauthor, encouraging office-mate, patient teacher, and great friend.

I appreciate the staff in the Computer Science Department, who have done so much to keep everything running smoothly and also build a sense of community, especially: Fay Alexander, Murray Anderegg, Robin Brennan, David Cowhig, Brandi Day, Jodie Gregoritsch, the late Bil Hays, Alicia Holtz, Denise Kenney, Beth Mayo, David Musick, Brett Piper, Gina Rozier, Mark Snyder, John Sopko, Mike Stone, Rosario Vila, Adia Ware, Missy Wood, and Hope Woodhouse. I would also like to thank all of the faculty, and especially Sanjoy Baruah, Gary Bishop, Fred Brooks, and Diane Pozefsky.

My time as a graduate student has been made more rich by multiple groups of people. I am thankful for the friendship and encouragement of everyone who has been part of the Graduate Women in Computer Science group, and especially for Tanya Amert, Bashima Islam, and Marie Nesfield. I appreciated serving

as a Computer Science Student Association officer along with Tanya Amert, Marc Eder, and Alan Kuntz. I have also enjoyed being part of both a reading group that has focused on Computer Science education and a Bible study with other computer scientists. I am thankful for the support of the UNC Writing Center and the encouragement from Nosh Mazandarani and the rest of the Accountabilibuddies group while writing my dissertation.

The work in this dissertation builds on knowledge and skills that so many people have taught me along the way. I am very thankful to have had so many wonderful teachers, coaches, instructors, and professors over the years.

I am endlessly grateful to my parents for their love and support. I am thankful for all of my family, and especially for the support of my brother, Colin, and all of my siblings-in-law: Amanda, Bradley, Margaret, Natalie, and Trey. I would also like to thank my friends who have encouraged me, especially the Kaufmann family.

Finally, I am grateful for Clay. He has encouraged me throughout the journey of graduate school, and without his love and support, this dissertation would not have been possible.

The research in this dissertation was funded by NSF grants CNS 1115284, CNS 1218693, CNS 1409175, CNS 1563845, CNS 1717589, CPS 1239135, CPS 1446631 CPS 1837337, CPS 2038855, and CPS 2038960, AFOSR grant FA9550-14-1-0161 ARO grants W911NF-14-1-0499, W911NF-17-1-0294, W911NF-20-1-0237, ONR grant N00014-20-1-2698, and funding from General Motors and FutureWei Corp. 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. This work was also supported by a Dissertation Completion Fellowship from the Graduate School at UNC.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

C-RNLP	Contention-Sensitive Real-Time Nested Locking Protocol
CGLP	Concurrency Group Locking Protocol
CPU	Central Processing Unit
DGL	Dynamic Group Lock
DPCP	Distributed Priority Ceiling Protocol
EDF	Earliest-Deadline First
fast RW-RNLP	Fast Reader/Writer Real-Time Nested Locking Protocol
FIFO	First-In, First-Out
FMLP	Flexible Multiprocessor Locking Protocol
FMLP+	Flexible Multiprocessor Locking Protocol (improved)
FP	Fixed Priority
G-C-RNLP	General Contention-Sensitive Real-Time Nested Locking Protocol
G-EDF	Global Earliest-Deadline First
GPU	Graphics Processing Unit
ILP	Integer Linear Program
L1	Level 1 (Cache)
L2	Level 2 (Cache)
L3	Level 3 (Cache)
M-BWI	Multiprocessor Bandwidth Inheritance Protocol
MCS	Mellor-Crummey and Scott Lock
MPCP	Multiprocessor Priority Ceiling Protocol
MrsP	Multiprocessor Resource Sharing Protocol
OMLP	$O(m)$ Locking Protocol
PCP	Priority Ceiling Protocol
PF-TL	Phase-Fair Ticket Lock
R^2LP	Reader-Reader Phase-Fair Locking Protocol
R^3LP	Reader-Reader-Reader Phase-Fair Locking Protocol
RCL	Remote Core Locking

R^kLP	k-Phased Reader-Reader Phase-Fair Locking Protocol
RM	Rate Monotonic
RNLP	Real-Time Nested Locking Protocol
RW-RNLP	Reader/Writer Real-Time Nested Locking Protocol
SUA	Schedulable Utilization Area
U-C-RNLP	Uniform Contention-Sensitive Real-Time Nested Locking Protocol

CHAPTER 1: INTRODUCTION

Real-time systems are distinguished by timing requirements in the form of task deadlines—the execution of processes (called *tasks*) within a *hard real-time system* must complete by their respective deadlines. As such, these systems differ from non-real-time ones in that they are evaluated on the basis of schedulability. *Schedulability* is the required guarantee that all deadlines will be met in any possible invocation pattern of the tasks in a system. The main focus of this requirement is *predictability*, in contrast to (perhaps more familiar) throughput-oriented metrics and notions of correctness; a missed deadline can result in significant financial loss and physical damage. Reasoning about schedulability requires considering sources of potential processing capacity loss, such as delays introduced by synchronization requirements caused by shared resources. The presentation and analysis of new approaches to handling shared resources can enable schedulability to be guaranteed for a broader class of systems.

1.1 Shared Resources

Tasks in a real-time system often require protected access to non-CPU components of the system called *shared resources*. (CPU access is granted by a separate mechanism, which is discussed in Chapter 2.) These components can range from physical elements (*e.g.*, a graphics processing unit) to data (*e.g.*, a region of shared memory).

Shared resources can be protected by locking protocols that guard access to each. For example, a locking protocol can ensure that each resource is used by at most one task at any given time. This behavior upholds the mutual-exclusion sharing paradigm. Without the appropriate protected access, a shared resource can enter an unsafe state. For example, a region of shared memory may hold inconsistent, incorrect values if two tasks attempt to update it concurrently.

Ensuring protected access by use of a locking protocol fundamentally introduces two types of delays. *Blocking* is the delay as a task waits for access to a resource (such as while another task is granted access). *Overhead* is the delay that occurs while a task waits for the protocol to determine if it may be granted access immediately or to update the protocol data structures that maintain the list of tasks awaiting access to include

this task. Thus, when examining a specific protocol, blocking is caused by the access order that the protocol specifies, and overhead is caused by how long it takes to manage the internal data structures that implement that ordering.

The choice of synchronization method for use in a given system is crucial, as excessive delays can cause missed deadlines and system failure. Schedulability is analyzed by determining whether there is sufficient processing capacity in the system to support all tasks meeting their respective deadlines. As such, the delays tasks can incur due to sharing resources with other tasks effectively reduces the available processing capacity; each task must be guaranteed to be able to incur these delays and still meet its deadline. A *schedulability test* is applied to a task system to determine if all task deadlines can be guaranteed to be met. The analysis of a task system with a schedulability test must incorporate blocking and overhead delays. Significant processing capacity can be lost to these delays (especially blocking), even if shared-resource access is rare at runtime. This is because, in schedulability analysis, *worst-case* behavior must be assumed.

1.2 Nested Resource Access

Tasks may require access to multiple resources concurrently. Multi-resource access is also called *nested resource access*. This term reflects the traditional manner in which resources are acquired, with one request for access "nested" within another.

For example, multi-resource access could be used to transfer a value from one data structure to another while maintaining a consistent value. Recent automotive industry specifications require an implementation for nested resource access (AUTOSAR, 2019). Additionally, nested resource access occurs in current systems, including the Linux kernel (Brandenburg and Anderson, 2007) and is reflected in benchmarks (Bacon et al., 1998).

Unfortunately, allowing nested resource access can result in excessive blocking caused by *transitive blocking chains*, described next. This further reduces system processing capacity.

1.3 Transitive Blocking Chain Problem

Transitive blocking chains can cause resource-requesting tasks, including those that do not require nested resource access, to be blocked even when the resources they have requested are free. Because worst-case

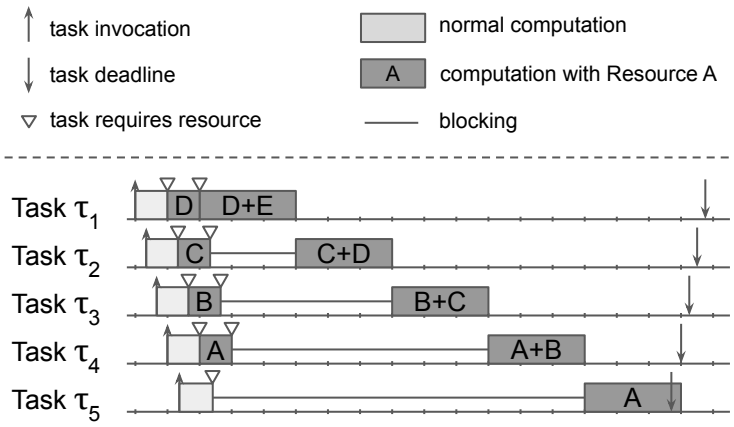


Figure 1.1: Possible execution pattern.

bounds must be applied in schedulability analysis, transitive blocking can be a key source of pessimism in multiprocessor schedulability analysis.

Example 1.1. Consider the five tasks depicted in Figure 1.1. Each task executes on a different core and most require access to two resources. For example, task τ_1 requires Resource D after executing for one time unit and then also requires Resource E after an additional time unit of execution. The illustrated execution pattern exhibits the worst-case scenario of transitive blocking for τ_5. It is blocked for over eleven time units, waiting for access to Resource A, which is not being used for the majority of that time. τ_5 is transitively blocked by all of the other tasks, though its resource requirements overlap with only one them. This blocking chain causes τ_5 to miss its deadline. ◊

As shown in Example 1.1, a blocking chain can build such that all tasks are transitively waiting on a single task. One might mistakenly assume that because τ_5 shares resources with only one other task, it will be blocked by at most one task execution. Instead, significant delays can be caused by the formation of a transitive blocking chain.

These blocking chains delay non-nested resource access as well as nested resource access. This highlights that the large blocking bounds caused by nested resource access can affect other types of access. As such, resource access cannot be analyzed in isolation, and the possibility of nested resource access by some tasks can impact every task.

The tasks described in Example 1.1 require mutually exclusive resource access (also called *write* access). However, some resources can also be shared with *read* access, in which multiple tasks can concurrently observe a shared resource without modifying it. Because read access can occur concurrently, one goal for

reader/writing locking protocols is to leverage this concurrent execution and grant read access with minimal blocking and protocol overhead.

In Example 1.1, if τ_5 required read access to Resource A, it could still experience the long waiting time caused by the transitive blocking chain under the protocol illustrated in Figure 1.1. The fundamental problem here is that access to Resource A is granted to task τ_4 for the eight time units that τ_4 is blocked on Resource B, and it may be unsafe to revoke τ_4's access to Resource A. Thus, without a protocol that fundamentally restructures access order or artificially delays access, such long blocking is inevitable.

In summary, the transitive blocking chains caused by nested resource access can delay all types of resource access, causing capacity loss and potentially compromising schedulability.

1.4 Thesis Statement

The focus of this dissertation is spin-based, fine-grained locking protocols that support nested resource access in hard real-time systems. A protocol is *spin-based* (as opposed to suspension-based) if a task waiting for resource access continues to execute on the CPU rather than yielding the CPU to another task. Additionally, this dissertation assumes non-preemptive execution from the moment any resource is requested until the moment when all shared resources have been released. A locking protocol in *fine-grained* if access is granted on a per-resource basis. This is in contrast to the coarse-grained approaches described in Chapter 2.

As illustrated above, by simply allowing nested resource access, significant delays can be caused, leading to lost processing capacity. Incorporating excessive delays into the schedulability analysis can reveal that a task system may not be guaranteed to meet all task deadlines. However, some system specifications require support of nested resource access. The two types of delays, blocking and overhead, are interdependent; a protocol with a more complex ordering for access satisfaction may reduce worst-case blocking at the cost of increased overhead. Balancing these concerns is crucial to ensuring schedulability.

Safely supporting nested resource access is non-trivial. The modifications to or the creation of many existing protocols to handle nested resource access has focused on safely sharing these resources. For example, both deadlock and starvation must be prevented. Deadlock can occur when two tasks each hold a resource that the other requires. Forward progress cannot be made without prevention of this scenario or intervention to force one of the tasks to lose resource access prematurely. Starvation, in which a task waiting

for access to a resource is never granted access, can occur if no access order is enforced by the protocol. For example, if the access order is random, a given task may never be guaranteed to be granted resource access.

Given the challenge of ensuring safe nested resource access, most existing work has been developed with that goal at the forefront, without focusing on reducing blocking. However, as illustrated in Example 1.1, without considering the resulting blocking as a primary concern, significant blocking can be caused.

Excessive blocking can cause tasks to miss their deadlines, resulting in an unsafe system. A simple locking protocol comprised of only first-in-first-out (FIFO) queues results in worst-case blocking exponential in nesting depth (Takada and Sakamura, 1995). More recent work has presented the RNLP family of protocols (Ward, 2016; Ward and Anderson, 2012, 2013, 2014), which was designed specifically to bound blocking to the number of processors in the case of the spin-based variant.

Blocking can be quantified on a per-request basis. For example, in an m-core system with no nesting and non-preemptive execution of critical sections, a fine-grained FIFO protocol yields $O(m)$ blocking for a given request. Blocking can also be measured relative to the *contention* a request experiences, the number of other requests with which it shares an overlapping set of resources. A protocol is *contention-sensitive* if the worst-case blocking is upper-bounded by a constant factor of each request's contention. Contention-sensitive blocking is the ideal, as it ensures blocking is at most (a constant factor larger than) the number of requests that require the same resource(s). It is not possible to achieve better blocking for all requests. Contention-sensitivity serves as the underlying motivation at the heart of much of this work.

Reducing interference (blocking and overhead) is crucial to ensuring that deadlines of systems with nested resource sharing can be met. In addition to evaluation on the basis of blocking and overhead, new approaches are evaluated on schedulability by conducting a *schedulability study*, which is the application of a schedulability test to a wide range of synthetic task systems with varying parameters. If new approaches reduce overhead, lower blocking, and outperform prior approaches in a schedulability study, that indicates that systems that could previously not be guaranteed to be schedulable can now have that guarantee. The work contained in this dissertation was primarily motivated by a single question: can contention-sensitive blocking (*i.e.*, blocking that is dependent only on tasks that access the same resource) be achieved with low overhead?

This leads to the following thesis statement:

New approaches to supporting nested resource access can mitigate or eliminate the transitive blocking chains that could otherwise increase blocking, without significant overhead. This can be accomplished with new locking protocols that isolate non-nested resource requests and read requests, protocols that reorder the satisfaction of nested write requests, and a protocol-independent approach to reducing overhead. These improvements in turn allow for schedulability to be confirmed for more task systems.

1.5 Contributions

The thesis is supported by the following contributions.

1.5.1 The Fast RW-RNLP

A primary problem of nested resource access is its impact on tasks requiring only non-nested or read access to shared resources. These types of access can be granted without forming transitive blocking chains, but can be significantly delayed by nested resource access.

To reduce these impacts, read and non-nested write access can be isolated by the use of a novel fast-path protocol. This protocol, the fast Reader/Writer Real-Time Nested Locking Protocol (fast RW-RNLP), is presented in Chapter 3. The fast RW-RNLP is a modular protocol that uses separate components for each access type (nested write, non-nested write, and read access). This separation removes negative interactions between requests for different access types.

The use of access-type-specific components necessitates the use of an arbitration mechanism to coordinate access between components. Two such arbitration mechanisms are presented in Chapter 3. The presented fine-grained arbitration mechanism operates on a per-resource basis. The pathological cases that result in unacceptable blocking with nested resource access are obviated by allowing only one request per resource per type to compete for access at a time. The coarse-grained arbitration mechanism instead cycles through the types, allowing one request per resource to execute during each type phase.

Chapter 3 also presents an evaluation of the fast RW-RNLP with both arbitration mechanisms. First, the overhead of each protocol is compared to prior work, and then a large-scale schedulability study is presented.

1.5.2 The C-RNLP

As described above, nested resource access can cause excessive blocking, building blocking chains between otherwise independent tasks. The Contention-sensitive Real-time Nested Locking Protocol (C-RNLP) is instead designed to be contention-sensitive: the blocking a task experiences is proportional to the number of other tasks that require an overlapping set of resources.

The C-RNLP, presented in Chapter 4, builds on an existing protocol, but instead allows a new request for resource access to cut ahead of existing requests if no already-waiting request will be delayed as a result. This runtime check is necessary to prevent starvation, but it requires maintaining data on each active resource request and adds overhead as a result. The cutting-ahead mechanism, however, enables a significant reduction in blocking: resource requests can skip ahead, filling gaps in the satisfaction order and preventing the formation of transitive blocking chains. Chapter 4 presents a study that examines the schedulability benefit of the reduced blocking at the cost of higher overhead.

1.5.3 The CGLP

While the C-RNLP improves blocking, it does so at the expense of higher protocol overhead. This prompts reframing the challenge of granting nested resource access and focusing on a solution with low overhead.

The Concurrency Group Locking Protocol (CGLP) is motivated by shifting from viewing a locking protocol as merely preventing resources from being accessed concurrently to instead viewing it as a mechanism that safely allows concurrency with respect to shared resources. The CGLP forms concurrency groups—sets of tasks that can safely execute concurrently (*i.e.*, without requiring overlapping resources)—and arbitrates among them. Chapter 4 presents multiple methods of forming the concurrency groups. For example, one of these methods translates the resource requirements into a graph coloring problem in order to determine which requests can execute concurrently.

The CGLP tackles the challenges of efficiency in blocking and overhead together by leveraging offline computing capacity to determine the concurrency groups. Offline group formation allows a simpler locking mechanism to be employed, resulting in lower runtime overhead.

Chapter 4 shows that concurrency groups can be formed with an integer linear programming approach in a reasonable amount of time offline. Then, a large-scale schedulability study is presented, which demonstrates the schedulability benefits of the reduced blocking and overhead from the CGLP.

1.5.4 Lock Servers

As described briefly above, some protocols that reduce blocking, such as the C-RNLP, do so by maintaining extensive data compared to traditional approaches. Access to this data, which is part of the state of the locking protocol, can cause significant delays. Use of the locking protocol updates this state, invalidating cached state. As such, maintaining resource request data can cause significant protocol overhead.

Lock servers are presented in Chapter 5 as a method to reduce protocol overhead. A *lock server* is a process that executes the locking protocol on behalf of resource-requiring tasks. This allows the protocol state to remain cache-local for the lock server. Chapter 5 presents four possible lock server configurations. These configurations can reflect the underlying cache structure and coordinate access between subsets of the task system.

Lock servers can be applied to any protocol and can significantly reduce the protocol overhead by centralizing the lock state and taking advantage of the underlying machine architecture. An experimental evaluation shows that lock servers can significantly reduce protocol overhead.

1.6 Organization

The remainder of this document is organized as follows. Chapter 2 presents terminology, notation, and related work. Chapter 3 proposes a modular protocol, the fast RW-RNLP, that significantly reduces synchronization delays for non-nested and read access of shared resources. Two protocols that reduce delays for nested resource access, the C-RNLP and the CGLP, are presented in Chapter 4. Both protocols build on notions of reordering request satisfaction. Chapter 5 gives a protocol-independent method of reducing overhead. Chapter 6 summarizes the contributions and poses open problems for future work.

CHAPTER 2: BACKGROUND

This section introduces relevant background information and related notation. Additionally, prior work is discussed. First, the general task model is presented. Then, multiple scheduling algorithms are given, followed by a discussion of schedulability analysis and a description of the most relevant approaches. Next, the resource model is given along with a classification of resource access types and a summary of methods for managing nested resource access. Then, metrics for the evaluation of synchronization protocols are presented along with a discussion of how the protocol analysis is incorporated into schedulability analysis. The chapter concludes with a summary of the most relevant locking protocols.

2.1 Task Model

This dissertation focuses on real-time systems that are comprised of recurrent programs called *tasks* on a multiprocessor platform with m processors. An arbitrary task is denoted τ_i; the task system consists of n tasks and is defined as the set $\Gamma = \{\tau_1, \ldots, \tau_n\}$. Each invocation of τ_i is called a *job*. The j^{th} job of τ_i is denoted $J_{i,j}$. (An arbitrary job of τ_i is denoted simply J_i.) Each job of τ_i is *invoked* with some minimum separation, which is the *period* of that task, denoted T_i; this corresponds to the classic sporadic task model (Mok, 1983). (The sporadic model extends the periodic model, in which job invocations are separated by exactly the period interval.) The *relative deadline*, denoted d_i, of a task τ_i is the span of time in which one of its jobs must *complete* after its invocation (also called its *arrival*). If the relative deadline of a task equals its period, it has an *implicit deadline*. A task has a *constrained deadline* if its relative deadline is at most its period, and an *arbitrary deadline* otherwise. Implicit-deadline tasks are assumed in this dissertation. The *absolute deadline* of a job is the point in time by which its execution must complete. The *worst-case execution time* of a task τ_i is denoted C_i. In hard real-time systems, the focus of this dissertation, all task deadlines must be met; the execution of each job must complete before its absolute deadline.

The *utilization* of τ_i is denoted u_i and is defined as $u_i = \frac{C_i}{T_i}$. The *system utilization* is denoted U and is the summed utilization of all tasks. Thus, $U = \sum_{j=1}^{n} u_j$. The *density* of a task τ_i is denoted δ_i and is defined as $\delta_i = \frac{C_i}{\min(T_i,\, d_i)}$.

2.2 Scheduling Algorithms

A task system is scheduled on a multiprocessor system by a *scheduling algorithm*. The discussion in this section begins with uniprocessor scheduling algorithms.

A job is *ready* to be scheduled from the time it is invoked until it has completed its execution. The job allowed to execute on a CPU at a given time is determined by the scheduling algorithm in use by the operating system. This dissertation focuses on systems in which the job to schedule on a CPU is chosen at runtime based on the priority of ready jobs. (This is in contrast to systems that use fixed schedules that are determined offline, like with the Cyclic Executive scheduler (Baker and Shaw, 1988, 1989).)

With priority-based scheduling algorithms, the highest-priority ready job is allowed to execute. The differences between these scheduling algorithms are in how priorities are assigned. (Under any approach, if multiple jobs share the highest priority, some tie-breaking mechanism is required, such as breaking ties in favor of the job with the lowest task ID.)

In particular, this work assumes the use of a *job-level fixed-priority* scheduling algorithm—the priority of a given job will not change over time. The following two task-level priority definitions both result in job-level fixed-priority assignments.

Fixed-priority (FP) scheduling algorithms use priorities assigned on a per-task basis prior to runtime—each job of a given task will have the same priority. For example, the *rate monotonic (RM)* scheduling algorithm (Liu and Layland, 1973) assigns priorities in order of period length; the task with the shortest period has the highest priority. Similarly, the *deadline monotonic* scheduling algorithms (Leung and Whitehead, 1982) assigns priorities based on relative deadline, with the highest priority being assigned to the task with the smallest relative deadline.

Dynamic-priority scheduling algorithms instead assign priorities on a per-job basis. For example, the *Earliest-Deadline-First (EDF)* scheduling algorithm (Liu and Layland, 1973) assigns priorities based on absolute deadlines: the highest priority is assigned to the job with the earliest deadline.

Both fixed-priority schedulers and dynamic-priority schedulers fall under the broader category of job-level fixed-priority schedulers, as the priority of a job relative to other jobs does not change during its execution under either type of scheduling algorithm.

Scheduling algorithms can also be distinguished by preemptivity: a *non-preemptive* scheduling algorithm ensures that a job continuously executes without interruption until completion. In contrast, *preemptive*

schedulers consider each newly invoked job and may allow such a job to preempt an executing job before it has completed (if it is not in a non-preemptive region). This dissertation focuses on the evaluation of systems that use preemptive scheduling algorithms.

The above scheduling algorithms were developed for uniprocessor systems and have been extended to multiprocessor systems in multiple ways. A multiprocessor system can employ *global*, *clustered*, or *partitioned* scheduling. In a globally scheduled system, each task is allowed to execute on any of the m processors. At any given time, all ready jobs are considered, and the (up to) m ready jobs with the highest priorities are granted a processor. For example, the *Global Earliest-Deadline First (G-EDF)* scheduling algorithm allows up to m ready jobs with the earliest absolute deadlines to execute on the m processors. A partitioned system is scheduled on a per-processor basis; each task is assigned to a processor offline, and scheduling decisions are then made on each processor using a uniprocessor scheduling algorithm. For a clustered system, the task system is split into a pre-determined number of clusters, each of which are scheduled on a subset of the processors. As such, a clustered system can be seen as a generalization of global and partitioned systems (with 1 and m clusters, respectively). More nuanced approaches have also been considered (Baruah and Brandenburg, 2013; Bastoni et al., 2011; Bonifaci et al., 2016, 2017; Cerqueira et al., 2014; Tang and Anderson, 2020; Voronov and Anderson, 2018). These include semi-partitioned systems (Anderson et al., 2005), which assign most tasks to a single processor, but allow a subset of tasks to migrate. A generalization of this approach instead allows the assignment of processor affinities, which specify a subset of the processors on which execution is allowed on a per-task basis.

2.3 Schedulability Analysis

A task set is *schedulable* if it can be guaranteed that all task deadlines will be met under the chosen scheduling algorithm. A *schedulability test* yields a conclusion about whether the task system can be guaranteed to meet all job deadlines under a given scheduling algorithm. A significant amount of research has presented a range of tests for both uniprocessor and multiprocessor scheduling algorithms.

In the following discussion of classifications of schedulability tests, the utilization-based test of $U \leq 1$ for a uniprocessor system scheduled with EDF (Liu and Layland, 1973) will be used as an example. For the system being analyzed, if $U > 1$, then it fails the utilization test. If instead $U \leq 1$, then the system passes the test. Any test can be categorized by the implication of its result for the system to which it is applied.

A test is *necessary* if failure of the test implies that the system cannot be scheduled. The test of $U \leq 1$ is necessary for all uniprocessor systems, regardless of the scheduler employed (Liu and Layland, 1973). Intuitively, with a workload (U) larger than the capacity (1 for a uniprocessor), deadlines will be missed. A schedulability test is *sufficient* if passing the test implies that the system to which it is applied is schedulable. For a uniprocessor system scheduled with EDF, the test $U \leq 1$ is sufficient (Liu and Layland, 1973). A schedulability test is *exact* if it is both necessary and sufficient. For uniprocessor analysis, the test $U \leq 1$ is exact for EDF (assuming implicit deadlines): a task system is schedulable with EDF if and only if $U \leq 1$.

The synchronization approaches presented in this dissertation can be applied to any system scheduled with a job-level fixed-priority scheduler. However, in the evaluations in this dissertation, the use of G-EDF is assumed and thus its analysis is the focus of the remainder of this section.

Prior work (Bertogna and Baruah, 2011) has summarized schedulability tests for G-EDF. Most of these tests are incomparable, so this work also suggests a combined approach to determining schedulability, in which a fast-running test is conducted first, before attempting slower, but possibly better-performing, tests.

Recall that if a sufficient but not necessary test fails, no conclusion can be drawn about schedulability. Instead, the application of a different sufficient test may reveal that the task system is indeed schedulable.

A short summary of each of the tests used for G-EDF in this dissertation is given here. These tests are applied in succession. Either failing a necessary test or passing a sufficient test terminates the application of the sequence of tests, as schedulability has been determined. Conversely, passing a necessary test or failing a sufficient test simply results in moving on to the next test. If, after the application of all tests, no answer has been found, the task system is reported as not schedulable, as it could not be shown to be schedulable. As described in prior work (Bertogna and Baruah, 2011), these tests have varying runtime costs, which influences the order in which they are chosen to be applied.

First, a simple necessary test of the density is applied for each task ($\forall \tau_i : \delta_i \leq 1$): if any task's execution time exceeds its relative deadline or period, it cannot be scheduled. Next, the necessary utilization test $U \leq m$ is applied. Then, a sufficient window-based test, BAK (Baker, 2003, 2005), is checked. BAK is derived by negating necessary conditions on the workload in a window of time preceding a missed deadline. Next, a necessary density-based test, GFB (Goossens et al., 2003), is applied. GFB accounts for the total density of the system and the maximum density of any task. Then, a second window-based test, BAR (Baruah, 2007), is applied. For tests based on the workload during an interval of time, *carry-in work*, the execution of jobs invoked before the window, must also be accounted for. In contrast to prior window-based schedulability tests

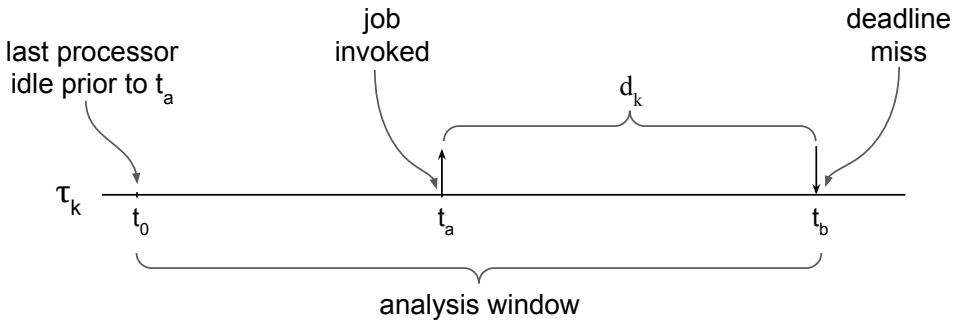


Figure 2.1: Illustration of analysis window for BAR schedulability test. Figure adopted from original presentation (Baruah, 2007).

for G-EDF that bound tasks contributing carry-in work by n (*e.g.*, BAK and BCL (Bertogna et al., 2005)), BAR considers a slightly different window and bounds the number of tasks contributing carry-in work by $m-1$.

As BAR is the final test to be applied, it is explained in slightly more detail. This test is derived by first considering the necessary conditions for an arbitrary task τ_k to miss a deadline by supposing that one of its jobs misses a deadline at time t_d. This job was invoked at $t_a = t_d - d_k$. BAR considers a time interval beginning at the latest instant in time at which at least one processor was idle, denoted t_0. The term $I(\tau_i)$ denotes the work done by jobs of τ_i over the interval $[t_0, t_d)$ during which each of the m processors is busy with jobs other than the job of τ_k that misses its deadline. This analysis window is illustrated in Figure 2.1. Between t_a and t_d, the job of τ_k that missed its deadline was able to execute for less than C_k. Thus, $\sum_{\tau_i \in \Gamma} I(\tau_i) > m \cdot (d_k - C_k + (t_a - t_0))$. The sufficient schedulability test BAR is derived by showing that for all tasks, this inequality cannot be satisfied. This is accomplished by also considering the potential for carry-in jobs (jobs invoked before t_0 that contribute to I for that task) (Baruah, 2007).

2.4 Resource Model

When a job requires access to a shared resource, it *issues* a request. An arbitrary request is denoted $\mathcal{R}_i$, and an arbitrary resource is denoted ℓ_a. There are n_r shared resources in the system. A request is *satisfied* once it (and thus, its issuing job) is granted access to the required resource(s), which the request *holds* until it *completes* (when the job no longer requires access) and *releases* the resource(s). A request is *active* from the time it is issued until it completes. A job is said to be within a *critical section* while it holds a resource. Thus, each request corresponds to a critical section. The maximum *critical-section length* of $\mathcal{R}_i$ is denoted L_i, and

the maximum critical-section length of any request is L_{max}. As described below, a request may be issued for one or more resources; the set of resources required by $\mathcal{R}_i$ is denoted $\mathcal{D}_i$. The *contention* of $\mathcal{R}_i$, denoted N_i, is the maximum number of active requests for an overlapping set of resources when $\mathcal{R}_i$ is issued.

2.4.1 Types of Resource Access

Requests can be distinguished by the type of access required. $\mathcal{R}_i$ is a *write request* if it requires mutually exclusive (write) access to $\mathcal{D}_i$ or a *read request* if other requests may access $\mathcal{D}_i$ concurrently with $\mathcal{R}_i$. When the distinction of request types is important, a write request is denoted $\mathcal{R}_i^w$ and a read request $\mathcal{R}_i^r$. If the type of a request is not specified, it is assumed to be a write request.

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 $\mathcal{D}_i$. In this case, $\mathcal{D}_i^w$ is used to denote the set of resources for which write access is required.

Additionally, some jobs require nested resource access, which occurs when a job requires access to multiple resources simultaneously. Requests are assumed to be *properly nested*; all resources acquired while a given resource is held are released before or concurrently with the release of that resource. The first request issued in a series of nested requests is called an *outermost request*. Correspondingly, the *outermost critical section* is the critical section corresponding to the first resource acquired (and thus the last released). Any request $\mathcal{R}_i$ in a series of nested requests is denoted $\mathcal{R}_i^n$, whereas a non-nested request $\mathcal{R}_i$ is denoted $\mathcal{R}_i^{nn}$. These superscripts can be combined with those denoting read or write requests. For example, a request $\mathcal{R}_i$ that is a nested write request is denoted $\mathcal{R}_i^{w,n}$. As with the other superscripts and subscripts, the distinction of nestedness is dropped when not relevant to a particular discussion. A summary of the notation is given in Table 2.1.

2.4.2 General Methods for Handling Nested Resource Access

Existing synchronization protocols tend to employ one of three primary methods to handle nested resource access. Recall from Section 1.4 that a locking protocol is fine-grained if each resource is protected individually.

To demonstrate the different approaches, the following running example is used.

Term	Explanation
m	Number of processors
τ_i	Task i
n	Number of tasks
Γ	Task set
$J_{i,j}$	j^{th} job of τ_i
J_i	Arbitrary job of τ_i
T_i	τ_i's period
C_i	τ_i's worst-case execution time
d_i	τ_i's relative deadline
u_i	Utilization of τ_i
U	Total system utilization
δ_i	Density of τ_i
$\mathcal{R}_i$	Request i
ℓ_a	Resource a
n_r	Number of shared resources
L_i	Maximum duration of $\mathcal{R}_i$
L_{max}	Maximum critical-section length
$\mathcal{D}_i$	Set of resources required by $\mathcal{R}_i$
N_i	Contention for $\mathcal{R}_i$
$\mathcal{R}^n$	Nested request
$\mathcal{R}^{nn}$	Non-nested request
$\mathcal{R}^w$	Write request
$\mathcal{R}^r$	Read request
$\mathcal{R}^{w,n}$	Nested write request
$\mathcal{R}^{w,nn}$	Non-nested write request
$\mathcal{R}^{r,n}$	Nested read request
$\mathcal{R}^{r,nn}$	Non-nested read request

Table 2.1: Notation.

Example 2.1. Consider the four jobs depicted in Figure 2.2. A snippet of pseudocode is shown for each, indicating which resources are required. For example, J_1 requires resource ℓ_a and then, depending on some expression x, may require ℓ_b. The request for ℓ_b would thus be nested within the request for ℓ_a. $\Diamond$

A key concern with nested resource access is that deadlock can be caused. *Deadlock* occurs when two or more tasks cannot make progress due to waiting for each other. With locking protocols, this occurs when unsatisfied requests will never become satisfied due to the resources currently held and the waiting resource requests.

Example 2.1 (continued). Observe that, as originally written, J_1 and J_2 could cause deadlock. Suppose J_1 acquires ℓ_a and J_2 acquires ℓ_b. Then, if J_1 requires ℓ_b, it must wait until ℓ_b is released by J_2. However, J_2

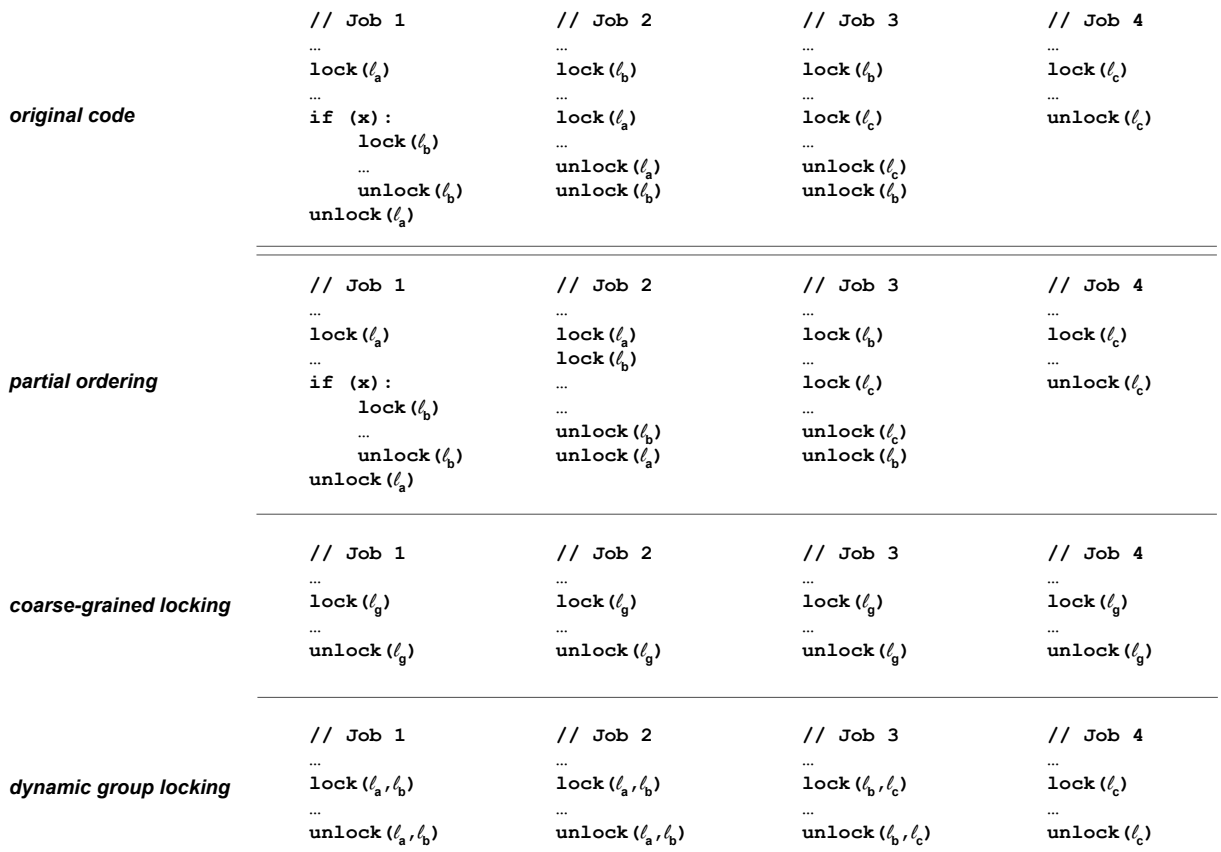


Figure 2.2: Illustration of the modifications for each of the three methods of handling nesting.

must similarly wait for the release of ℓ_a. With no intervention in this situation, deadlock occurs, and neither task can make progress. ◊

The following approaches present different methods for preventing deadlock.

2.4.2.1 Partial Ordering of Resources

Frequently, nesting is realized by allowing jobs to request and acquire each resource individually in a sequential fashion. Defining a partial order on the resources prevents deadlock if all resource acquisitions respect this ordering (Dijkstra, 1978; Havender, 1968).

Example 2.1 (continued). As depicted in Figure 2.2, the partial ordering selected for this system is alphabetical (once a resource has been acquired, only resources with identifiers later in the alphabet may be acquired). As such, J_2 must be modified to first acquire ℓ_a, and then ℓ_b. Any portion of J_2's execution which originally required only ℓ_b can now occur after both resources have been acquired. ◊

This approach can result in long transitive blocking chains like the one described in Chapter 1. Additionally, the individual acquisition of resources can significantly inflate the duration of a critical section, as the outermost request is satisfied (and then the corresponding resource is held) while waiting for satisfaction of a nested request. For instance, the blocking a job may experience while waiting to be granted access to a second resource must be counted toward the critical-section length of the outermost request (illustrated in Example 1.1), leading to delays exponential in nesting depth (Takada and Sakamura, 1995).

Some protocols (Takada and Sakamura, 1995; Ward, 2016; Ward and Anderson, 2012, 2013, 2014) apply an approach very similar to this, but instead apply a time-stamping mechanism and delay the execution of some requests in order to avoid potential deadlock scenarios. These approaches are protocol-specific, rather than general approaches that are applied broadly.

2.4.2.2 Coarse-Grained Locking

Redefining the granularity of resources is an alternate approach to ensuring safe (deadlock-free) resource access. By statically grouping the original set of resources into newly defined resources that represent one or more of the original resources, nesting can be eliminated.

Example 2.1 (continued). In Figure 2.2, all resources have been redefined to be in the static group now represented by ℓ_g. Thus, each job must acquire ℓ_g in order to proceed with its execution. ◊

Non-nested locking protocols can be directly applied to these newly defined resources. While this simple approach removes the possibility of deadlock, it may also remove a significant amount of possible parallel execution from the system, delaying the execution of some tasks and artificially reducing the number of systems that are schedulable. Indeed, in the worst-case, all resources may be reduced to a single redefined resource.

2.4.2.3 Dynamic Group Locking

The final common approach used by existing real-time nested locking protocols is to apply *dynamic group locks (DGLs)* (Ward, 2016; Ward and Anderson, 2013). To do so, all nested requests can be dynamically coalesced into a single request for all resources that may be required concurrently.

Example 2.1 (continued). This update is illustrated in Figure 2.2. Each task now issues only a single request in place of the previous outermost request. ◊

In the case of conditional code, applying DGLs may cause a job to acquire resource that is not necessary.

Example 2.1 (continued). Observe that, in Figure 2.2, J_1 issues a single request for both ℓ_a and ℓ_b under the DGL formulation; both resources must be acquired, even if the conditional would not have applied in this execution of the task. ◊

While this behavior may delay request satisfaction at runtime, this is not a significant detractor from the use of DGLs, as the analysis of such a task must consider that either resource is required anyway, already incorporating that pessimism.

DGLs provide a method of fine-grained locking, as each resource can be added to the dynamic group individually; this functionality is different from the coarse-grained locking described above, which is used to coordinate access to groups of resources that are statically determined offline.

Example 2.1 (continued). Note that τ_2 and τ_4 may execute concurrently, as they do not require any overlapping resources when DGLs are applied. This is in contrast to the single, statically defined shared resource required under the coarse-grained locking approach. ◊

With DGLs, each request can contain all resources that must be acquired in a nested fashion; once access is granted to all of these resources, the request will complete before the task requires access to more resources. As such, the applied locking protocol is given knowledge of the nested resource requirements, allowing deadlock prevention. (This knowledge of what would otherwise be "future" requests prevents the basic deadlock scenario, in which there is insufficient information in the locking protocol to prevent deadlock.)

To avoid critical-section inflation caused by resource ordering and lost parallelism caused by coarse-grained locking, this dissertation assumes the use of DGLs: each issued request is for all resources that may be required during the critical section of what would otherwise be the outermost request.

2.5 Metrics of Evaluation for Locking Protocols

Locking protocols are analyzed on the basis of overhead, blocking, and schedulability. All protocols can cause both blocking and overhead delays, and often make tradeoffs between the two; for example, a protocol that lowers blocking by reordering requests each time a new request is issued will likely have increased overhead. The blocking and overhead delays introduced by a synchronization protocol can in turn impact the schedulability of the system. These metrics of evaluation are discussed in more detail after some analysis assumptions are given.

2.5.1 Analysis Assumptions

For real-time systems, a common assumption is that possible requests from each task must be known *a priori*, as this information is required for determining schedulability.

Any locking protocol can cause a job to wait for resource access, and this dissertation assumes spin-based waiting (also called busy-waiting). This waiting is coupled with a *progress mechanism* (Brandenburg, 2011; Ward, 2016) that ensures a resource-holding task can complete its critical section and release its acquired resources. Here a busy-waiting spinlock is coupled with non-preemptive execution[1] for any active request.

As with prior work (Brandenburg, 2011; Ward, 2016), the number of critical sections and their durations are considered to be constants in the analysis of blocking. In this analysis, m and n are considered to be variables, as is the contention. Locking protocols can exhibit different scales of blocking. For example, a protocol that yields $O(m)$ blocking means that a single request may be blocked by $O(m)$ other requests (commonly one from each of the other $m-1$ cores in spin-based non-preemptive request execution). As described earlier, a contention-sensitive mutual-exclusion protocol achieves blocking asymptotically bounded by the number of conflicting requests.

Contention-sensitivity was described above for mutual-exclusion locking protocols. A reader/writer locking protocol ensures contention sensitivity for a request $\mathcal{R}_i$ if the worst-case blocking for $\mathcal{R}_i$ is $O(1)$ if it is a read request, and $O(N_i)$ if it is a write request.

2.5.2 Overhead

As mentioned in Chapter 1, locking protocols can cause a loss in processing capacity due to their underlying logic. A job may be delayed by the execution of the protocol to determine when its request will be satisfied; this delay can include the time required to update data structures required by the protocol, such as a queue of requests. Any delay introduced by the execution of the protocol is overhead.

2.5.3 Blocking

The focus of this dissertation is non-preemptive spin-based waiting. As such, *priority-inversion blocking* (pi-blocking) occurs when a job is prevented from executing by the execution of a lower-priority job. Thus, a

[1]The allowance of non-preemptive code sections is commonly assumed in real-time systems.

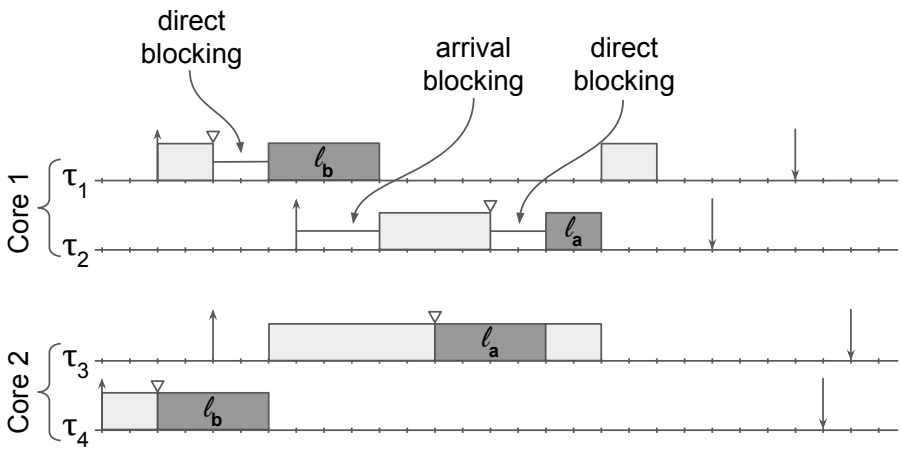


Figure 2.3: Illustration of invocation blocking and direct blocking.

job can be pi-blocked in the two scenarios described next. (Different pi-blocking definitions exist for systems in which waiting is realized by suspension (Brandenburg, 2011).)

One source of pi-blocking is the delay a job incurs due to waiting for resource access: *direct blocking* occurs when a job cannot execute because it has issued a request that is not yet satisfied due to the resource(s) currently being held by other job(s). When considering an individual request that is waiting for access, this waiting is referred to as *acquisition delay*. The ordering of request satisfaction by the protocol determines the magnitude of direct blocking.

Example 2.2. Consider the four tasks shown in Figure 2.3. Suppose a locking protocol is used that requires tasks with active resource requests to execute non-preemptively, in which tasks wait by busy-waiting. Direct blocking is depicted in Figure 2.3 when τ_1 is blocked while waiting for ℓ_b. ◊

Arrival blocking occurs when a job invocation is available but cannot execute because a lower-priority job is executing non-preemptively. This priority-inversion blocking is caused by the progress mechanism (here, non-preemptive execution), and is sometimes referred to as *progress-mechanism-related blocking*.

Example 2.2 (continued). In Figure 2.3, τ_2 is arrival blocked while τ_1 is executing its critical section non-preemptively. ◊

When the type is not specified, *blocking* refers to direct blocking. (The maximum arrival blocking a job may experience is asymptotically upper bounded by the maximum direct blocking of other jobs, so much of the analysis focuses on direct blocking.)

2.5.4 Schedulability Analysis of Locking Protocols

In order to test schedulability, blocking analysis for the chosen locking protocol must be conducted. This blocking analysis is then incorporated into schedulability analysis.

Schedulability can be assessed with well-known analysis techniques, like those described above for G-EDF, by inflating the execution time for each job by the blocking and overhead delays it could incur. This is a safe manner in which to account for non-preemptive regions (Brandenburg, 2011; Liu, 2000), though it can introduce additional pessimism. As discussed more fully in later chapters, there is often a tradeoff between blocking and overhead when comparing locking protocols; a more complex (higher overhead) protocol may be able to yield lower blocking. While blocking tends to be the dominant factor in schedulability analysis, accounting for overhead by inflating critical-section lengths magnifies the impact of overhead.

Within the context of protocols that grant access to resources in a non-nested fashion, computing reasonably tight blocking bounds has been an area of focus. For example, the impact of queue locks on low-priority tasks scheduled with a global fixed-priority scheduler has been explored to improve schedulability (Chang et al., 2010). Additionally, worst-case blocking bounds for a broad variety of lock types, including FIFO and priority-ordered, were tightened by using mixed-integer linear programming (Wieder and Brandenburg, 2013). More recently, a linear programming framework for analyzing blocking under global fixed-priority scheduling algorithms was presented and used to compare existing global semaphore protocols based on six types of delay (direct pi-blocking, indirect pi-blocking, preemption pi-blocking, regular interference, co-boosting interference, stalling interference) (Yang et al., 2015).

Challenges with analyzing arbitrary nesting have long been known. Computing a safe upper bound on blocking for a FIFO-based spinlock protocol that uses resource ordering results in a blocking bound exponential in nesting depth (Takada and Sakamura, 1995). Tightly bounding the blocking caused by arbitrarily nested resource access is NP-hard (Wieder and Brandenburg, 2014). A graph-based abstraction has been presented that serves as the basis for an integer linear program (ILP) to compute blocking for nested FIFO spinlocks (Biondi et al., 2016).

While the per-request inflation-based approach is safe, it can be pessimistic. More recently, inflation-free methods have been presented (Biondi and Brandenburg, 2016; Wieder and Brandenburg, 2013). These approaches capture the worst-case blocking delays that can accrue over time. In contrast to the inflation-based approach, these methods incorporate blocking directly within the schedulability-analysis framework, instead

of first inflating tasks. While these general frameworks can be extended, per-protocol analysis must first be developed.

2.6 Existing Approaches to Synchronization

The past three decades of research has resulted in a rich body of work on real-time locking protocols. This section summarizes the most relevant such approaches, and briefly discusses approaches beyond locking protocols for coordinating access to shared resources and protocols developed for use outside of real-time systems. For a more in-depth summary, see the recent systematic review (Brandenburg, 2019).

2.6.1 General Classifications

As described in Chapter 1, this dissertation focuses on the use of locking protocols to grant access to shared resources. Alternatives to locking protocols include retry-based (Anderson and Ramamurthy, 1996; Barros et al., 2015; Belwal and Cheng, 2011; Brandenburg et al., 2008; El-Shambakey, 2013; Ramamurthy, 1997; Sarni et al., 2009; Schoeberl and Hilber, 2010; Schoeberl et al., 2010; Yoo and Lee, 2008) and wait-free (Anderson et al., 1997; Brandenburg et al., 2008; Cho et al., 2007) approaches, which must be analyzed accordingly. For example, retry-based approaches require a job to repeatedly attempt to execute a critical section until it has safely completed. Instead of blocking analysis, such systems are analyzed by considering the maximum number of retries required.

In the remainder of this section, the most relevant locking protocols for non-nested and nested resource access are presented. Some of these approaches assume suspension-based waiting or allow preemptive spinning.

2.6.2 Locking Protocols for Non-Nested Resource Access

This section summarizes some key locking protocols that provide non-nested resource access. Some of these protocols have been developed specifically for real-time systems, while others were developed for more general systems and do not necessarily have accompanying real-time guarantees.

Recall from Section 2.4.2.2 that lock nesting can be trivially supported by redefining resources such that no nesting occurs by using a coarse-grained group lock. However, this can cause significant unnecessary blocking between tasks that do not actually share any resources. Thus, these approaches that do not handle

nested requests (except by means of a static, coarse-grained group lock) are discussed briefly before covering prior work that allows fine-grained lock nesting.

2.6.2.1 The MCS Lock

The MCS lock, named for its authors, is a mutual-exclusion lock that grants access in FIFO order (Mellor-Crummey and Scott, 1991a). The MCS lock is a queue-based lock. Each request busy-waits by spinning on a separate flag variable, enabling $O(1)$ memory references due to the issuance and completion of each request. Additionally, a very small amount of space is required for each lock. The MCS lock was developed to have very low overhead, which was achieved by the small memory footprint and minimal memory references. As such, it is often taken as the gold standard for a low-overhead protocol.

2.6.2.2 Real-Time Locking Protocols

There is a significant body of work on locking protocols for use in multiprocessor real-time systems. An overview of those that do not allow arbitrary nesting is given here, along with related work.

Several multiprocessor protocols expand on their uniprocessor counterparts. The distributed and multiprocessor priority ceiling protocols (DPCP and MPCP) (Rajkumar, 1990, 1991; Rajkumar et al., 1988) build on ideas of the priority ceiling protocol (PCP) (Sha et al., 1990). The MPCP was then used as a basis for developing a partitioning heuristic for resource-sharing tasks (Lakshmanan et al., 2009). The multiprocessor stack resource protocol expands on the ideas of the stack resource protocol (Baker, 1991) by classifying resources as local or global; access to global resources is coordinated in FIFO order, with waiting processes spinning non-preemptively (Gai et al., 2001). This was then compared to the MPCP (Gai et al., 2003). Beyond multiprocessor ceiling-based protocols (Chen and Tripathi, 1994), the FMLP (Block et al., 2007), FMLP+ (Brandenburg, 2014), and OMLP (Brandenburg and Anderson, 2013) have been developed. Many of the above protocols have been implemented and those implementations compared (Brandenburg, 2011; Brandenburg and Anderson, 2008a,b). While some of these approaches allow some amount of request nesting, such nesting is limited to only local resources or only coarse-grained resource groups. The focus of this dissertation is on supporting fine-grained, unrestricted nested resource access.

Prior work has also investigated different priorities at which a blocked task may spin; at lower priorities, spinning tasks may suspend (Afshar et al., 2014, 2017, 2018). Work has also been done to coordinate

resource sharing between independently developed system components that are then used modularly in a larger system (Afshar et al., 2013, 2015; Nemati et al., 2011).

2.6.2.3 Reader/Writer Locking Protocols

Under a reader/writer locking protocol, read requests may be satisfied concurrently with other read requests and mutual exclusion is enforced for write requests. Approaches to supporting reader/writer access include reader-preference (respectively, writer-preference) locks, which allow for starvation of write (respectively, read) requests (Courtois et al., 1971; Mellor-Crummey and Scott, 1991b). Another approach is *phase-fair* locking, in which read phases and write phases alternate (Brandenburg and Anderson, 2010).

The fundamental ideas behind phase-fair locks are crucial to the development of some of the protocols presented in this dissertation. As such, phase-fair reader/writer locking is described in detail here.

The phase-fair reader/writer lock is a non-preemptive spinlock. If both types of requests are active concurrently, the protocol alternates between *read phases* wherein read requests are given preference, and corresponding *write phases*.

At the start of a read phase for a resource, all active read requests for that resource are satisfied. Once those requests complete, a write phase may begin. (If a read request is issued during a read phase and there is a write request waiting, this newly issued read request must wait to be satisfied until the next read phase.) In a write phase, a single write request is satisfied. Write requests are satisfied in FIFO ordering relative to other active write requests. This specification ensures that each read request is blocked by at most one read phase and one write phase, and that each write request is blocked by at most a series of read and write phases that is bounded by the number of active write requests when that request was issued.

In Figure 2.4, gray shading indicates which requests will execute in the next phase.

Example 2.3. Figure 2.4 depicts the state of a phase-fair reader/writer lock at three time instants, t_1, t_2, and t_3, where $t_1 < t_2 < t_3$. At time t_1, four requests have been issued in increasing index order. Read request $\mathcal{R}_1^r$ is satisfied, and write request $\mathcal{R}_2^w$ will be satisfied after the read phase that includes $\mathcal{R}_1^r$ completes. $\mathcal{R}_3^r$ and $\mathcal{R}_4^w$ have also enqueued in their corresponding queues. At time t_2, $\mathcal{R}_2^w$ is satisfied, and an additional read request, $\mathcal{R}_5^r$, has been issued. In the next read phase, all read requests will be satisfied, as indicated by the gray shading. Indeed, at time t_3, $\mathcal{R}_3^r$ and $\mathcal{R}_5^r$ are satisfied. This read phase will be followed by a write phase, in which $\mathcal{R}_4^w$ will be satisfied. $\Diamond$

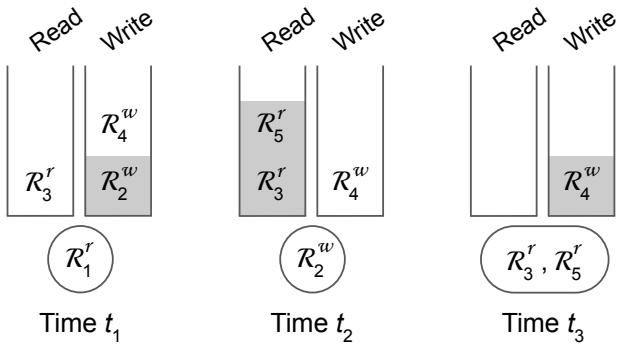


Figure 2.4: Illustration of phase-fair reader/writer locking protocol managing access to a resource.

Phase-fair reader/writer locks are perhaps the best contention-sensitive option in terms of lock/unlock costs (*i.e.*, the time required to acquire or release a lock) if all requests are non-nested requests (Brandenburg and Anderson, 2010). Several possible implementations of phase-fair locks were considered in prior work (Brandenburg and Anderson, 2010). Brandenburg and Anderson found the phase-fair ticket-lock (PF-TL) implementation to be comparable to or better than other phase-fair implementations from the perspective of lock/unlock costs.

2.6.2.4 Other Resource-Sharing Paradigms

Prior work has also explored resource-sharing schemes beyond mutual exclusion and reader/writer sharing. For example, k-exclusion locking protocols (Brandenburg and Anderson, 2011; Elliott and Anderson, 2013; Ward et al., 2012) allow up to k requests to access a shared resource concurrently. Also, new resource-sharing paradigms like preemptive mutual exclusion and half-protected sharing, which supports two request types (non-preemptive write requests and unprotected requests that are essentially preemptive read requests), have been developed (Ward, 2015).

2.6.3 Locking Protocols for Nested Resource Access

This section highlights the exiting approaches to supporting nested resource access in real-time systems.

2.6.3.1 Using Coarse-Grained Approaches

As described above, applying static group locks allows protocols designed for non-nested locking protocols to be applied to systems in which nested resource access is required. However, doing so necessitates

redefining the resources to coalesce all potentially overlapping resources into a group protected by a single lock. This grouping of resources eliminates the concurrency that would otherwise be possible with a fine-grained (per-resource) locking approach.

2.6.3.2 Early approaches

One early solution to the potential deadlock or long blocking caused by nested requests uses an online schedulability test to determine if resource access would be granted (Schwan and Zhou, 1992). If satisfying a request would cause a deadline miss (possibly due to either causing deadlock or simply excessively high blocking), the request is denied, and the job must instead handle that exception.

In addition to showing that unrestricted lock nesting can lead to blocking exponential in nesting depth, an early work presented a protocol that, for systems with nesting depth at most two, ensures $O(m)$ blocking (Takada and Sakamura, 1995). However, if nesting depth exceeds two, this protocol again results in exponential blocking.

A locking protocol that builds on the PCP achieves fine-grained locking by requiring all possibly required resources to be requested prior to executing the critical section (Rhee and Martin, 1995). While waiting for some of the required resources, a job may be required to release an already acquired resource to allow a higher priority job to execute. Unfortunately, this approach only applies to a partitioned system.

Both the blocking caused by nested resource requests and the preemption of resource-holding jobs can increase the blocking of waiting requests. To address this problem (but not the exponential blocking), two priority-inheritance-based spinlock algorithms for nested resource access were proposed (Wang et al., 1996).

2.6.3.3 M-BWI

The *multiprocessor bandwidth inheritance protocol (M-BWI)* (Faggioli et al., 2010, 2012) expands the original uniprocessor bandwidth inheritance protocol (Lipari et al., 2004), which uses a helping mechanism to allow a resource-holding job to execute. (A *helping mechanism* provides a means for job waiting for resource access to ensure progress for the resource-holding job.) The *bandwidth* of a task is a measure of what proportion of execution time on a CPU it is granted. This protocol enables a resource-holding job that would otherwise be suspended to instead continue executing by inheriting the bandwidth of a higher-priority

job. M-BWI accomplishes this by reasoning about resource reservation servers that grant bandwidth and coordinating between such servers on different processors.

While M-BWI enables sharing bandwidth in order to reduce the time that a resource is held by an unscheduled job, it does nothing to change from a FIFO order of request satisfaction. As such, long transitive blocking chains can form under M-BWI as described in Chapter 1, resulting in exponential blocking (Takada and Sakamura, 1995). Therefore, M-BWI cannot be considered a solution to the challenge of high blocking in the presence of nested requests.

2.6.3.4 MrsP

As with a couple of non-nested protocols, the *Multiprocessor Resource Sharing Protocol (MrsP)* (Burns and Wellings, 2013; Garrido et al., 2017; Zhao et al., 2017, 2020) builds upon the PCP by using the PCP locally to bound accesses to global resources. It then employs a helping mechanism in which a blocked task may allow a preempted task to execute. Nested resource access is allowed by MrsP, and the priority ceilings of some resources may be recalculated if a nested resource access occurs while that resource is held (Garrido et al., 2017). More recently, schedulability analysis has been refined for MrsP (Zhao et al., 2017, 2020). As with the M-BWI, MrsP requires a partial ordering on resources. This, along with the structure of protocol, results in analysis that again must incorporate the blocking time incurred by inner requests into the critical-section lengths or blocking for outer requests; the summed blocking plus critical-section length for each resource is defined recursively on the resource(s) last in the partial order (Garrido et al., 2017; Zhao et al., 2020). Similarly to prior approaches, MrsP does not propose methods to break the transitive blocking chains that can occur with nested resource access.

2.6.3.5 RNLP Family

The remaining protocols that support nested resource access are in the *real-time nested locking protocol (RNLP)* family of protocols. The RNLP (Ward, 2016; Ward and Anderson, 2012, 2013, 2014) is the first set of protocols to provide asymptotically optimal pi-blocking for nested requests. It does so by employing per-resource ordering and granting access on a timestamp-ordered basis.

The RNLP family of protocols allows waiting to be realized by spinning or suspension; different mechanisms are used for dealing with priority inversions depending on how tasks are scheduled. For the non-preemptive, spin-based variants, worst-case pi-blocking is $O(m)$ for write requests.

Variants on the original protocol include incorporating the use of DGLs (Ward and Anderson, 2013) and extending the approach to support reader/writer locking (Ward and Anderson, 2014). The reader/writer RNLP (RW-RNLP), like the phase-fair reader/writer locking protocols, coordinates between reader and writer phases on a per-resource basis.

The existing RNLP variants yield $O(m)$ worst-case blocking for write requests when nesting is allowed. While this is a significant improvement on prior approaches, the existing RNLP variants are not contention-sensitive, even for non-nested write requests. This dissertation presents new contention-sensitive protocols to the RNLP family.

2.6.4 Chapter Summary

This chapter has presented the task model and resource model assumed for the remainder of this dissertation. A brief introduction to scheduling algorithms was given, followed by a discussion of how those algorithms are analyzed for schedulability. Additional details were given for the analysis of systems scheduled with G-EDF, which are used in the evaluations in this dissertation. Existing locking protocols were discussed, along with the basis for the evaluation of protocols on overhead, blocking, and schedulability.

CHAPTER 3: MINIMIZING IMPACTS ON READ AND NON-NESTED WRITE REQUESTS[1]

Evidence suggests that while nested resource access does occur, non-nested resource access is more common (Bacon et al., 1998; Brandenburg and Anderson, 2007). When resource requests may be nested, the protocol used to control access must support nested requests. The only existing protocols that support arbitrary nesting and yield asymptotically optimal blocking bounds are those in the RNLP family.

Unfortunately, this support comes at the cost of higher overhead for read requests and non-nested write requests. Using a trivial solution like the resource-ordering approach illustrated in Example 1.1 can instead result in unacceptably high blocking, as discussed in Chapter 1. All existing protocols either yield non-contention-sensitive blocking for the most common resource access types or have increased overhead compared to approaches that do not support nesting. Thus, these protocols support the less common nested resource access to the detriment of more common resource access types. Though nested access may be rare in a system, that system must still be able to safely support such access patterns.

These observations motivate a new *fast-path mechanism* for the RNLP family. It was designed with the goal of ensuring that read access and non-nested write access both **(i)** are contention-sensitive and **(ii)** incur low lock/unlock overhead comparable to that of single-resource protocols. This new protocol in the RNLP family is called the *fast* RW-RNLP.[2] This chapter presents the fast RW-RNLP with two versions for one of the internal components: one, a protocol variant applied in a constrained setting called the RW-RNLP*, and the other, a reader-reader-reader phase-fair locking protocol called the R^3LP. This work builds directly on phase-fair ticket locks (PF-TLs) and the RW-RNLP, both of which are described in more detail in Chapter 2.

In the fast RW-RNLP, non-nested requests are immune from the effects of transitive blocking chains caused by nesting. This is achieved by employing a modular design that mostly separates concerns related to

[1]Contents of this chapter previously appeared in preliminary form in the following papers:

Nemitz, C., Amert, T., and Anderson, J. (2017). Real-time multiprocessor locks with nesting: Optimizing the common case. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*.

Nemitz, C., Amert, T., and Anderson, J. (2019a). Real-time multiprocessor locks with nesting: Optimizing the common case. *Real-Time Systems*, 55(2):296–348.

[2]The terminology "fast-*in-the-common-case* RW-RNLP," which is obviously too verbose, would be more technically precise.

handling nested and non-nested requests. When no nested requests occur, the fast RW-RNLP can function nearly identically to a set of per-resource PF-TLs, depending on the choice of one component.

This similarity to PF-TLs is reflected in experiments measuring lock/unlock overhead and observed pi-blocking times. The fast RW-RNLP is also evaluated and compared to other protocols in a large-scale schedulability study. The fast RW-RNLP variants, and the R^{3}LP in particular, tended to outperform the other protocols when non-nested requests are the common case.

This chapter begins with an introduction to reader-only phase-fair locking protocols, which are used in two components of the fast RW-RNLP. Next the structure of fast RW-RNLP is presented, followed by discussion of the R^{3}LP and the RW-RNLP*, along with blocking bounds for both protocol variants. Then, an evaluation of overhead, blocking, and schedulability is presented. The chapter concludes with some additional details, including showing that the previously given blocking analysis is tight.

3.1 Reader-Only Phase-Fair Locks

Two components of the fast RW-RNLP employ *reader-only phase-fair locks*, which were introduced in a restricted form to support the work presented in Chapter 5. Reader-only phase-fair locks build on the mechanisms of phase-fair reader/writer locks (Brandenburg and Anderson, 2010), covered in more detail in Chapter 2. In preparation for the presentation of the fast RW-RNLP, an implementation of and the bounds on worst-case acquisition delay for a phase-fair variant is presented. As in prior work (Ward and Anderson, 2014), it is assumed that all lock and unlock invocations take no time.

3.1.1 Reader-Reader Phase-Fair Locks

The reader-reader phase-fair locking protocol (R^{2}LP) arbitrates access to a resource between two types of read requests; an arbitrary read request of Type 1 (respectively, Type 2) may execute concurrently with requests of the same type but may not execute with requests of Type 2 (respectively, Type 1). (The problem of supporting multiple types of read requests is similar to the group mutual exclusion problem (Joung, 2000; Keane and Moir, 1999, 2001) except for the additional requirement of $O(1)$ pi-blocking bounds.)

A read request of Type 1 is denoted as $\mathcal{R}_i^{r1}$ and a read request of Type 2 as $\mathcal{R}_i^{r2}$. Under the R^{2}LP, requests enqueue in the "lane" corresponding to their type. For example, $\mathcal{R}_3^{r1}$ in Figure 3.1 is enqueued in Lane 1. The following example illustrates the phase transitions of the R^{2}LP.

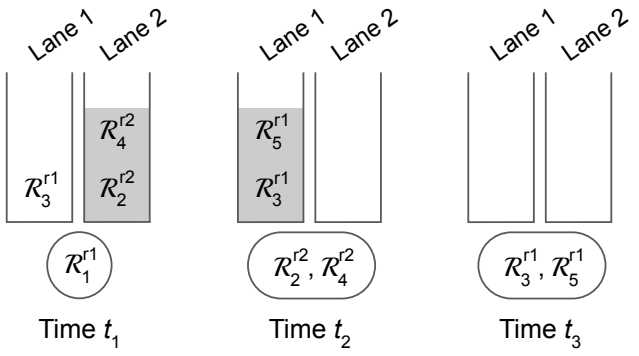


Figure 3.1: R^{2}LP illustration with read requests of Type 1 and Type 2.

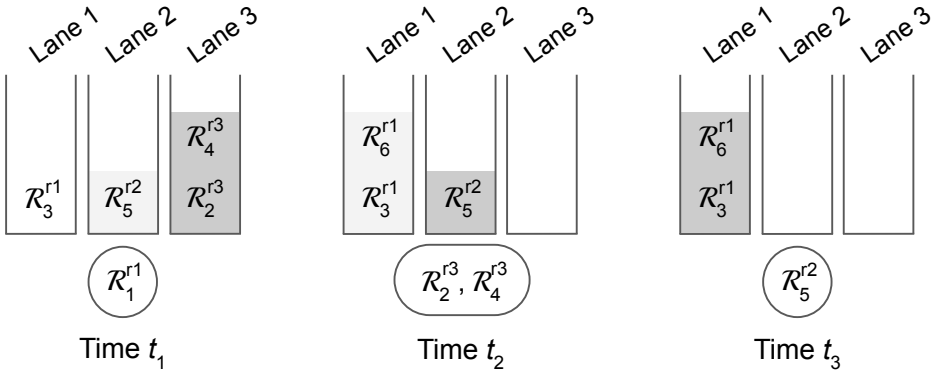


Figure 3.2: R^{3}LP illustration with read requests of Type 1, Type 2, and Type 3.

Example 3.1. As shown in Figure 3.1, at time t_1, a read request of Type 1, $\mathcal{R}_1^{r1}$, is satisfied. The group of requests in Lane 2, $\mathcal{R}_2^{r2}$ and $\mathcal{R}_4^{r2}$, will be satisfied in the next phase, as indicated by the gray shading. Though $\mathcal{R}_3^{r1}$ is of the same type as the satisfied request, it cannot be satisfied at t_1; allowing such behavior could cause starvation. Therefore, the R^{2}LP prevents this and instead allows requests of the other type to be satisfied after all currently satisfied requests complete.

At time t_2, $\mathcal{R}_1^{r1}$ has completed and $\mathcal{R}_2^{r2}$ and $\mathcal{R}_4^{r2}$ are satisfied. Additionally, $\mathcal{R}_5^{r1}$ has been issued. At time t_3, $\mathcal{R}_2^{r2}$ and $\mathcal{R}_4^{r2}$ have completed, and both $\mathcal{R}_3^{r1}$ and $\mathcal{R}_5^{r1}$ are satisfied. ◊

3.1.2 Reader-Reader-Reader Phase-Fair Locks

The reader-reader-reader phase-fair locking protocol (R^{3}LP) arbitrates resource access among three types of read requests; read requests of one type may run concurrently with other read requests of the same type but must be prevented from accessing the resource concurrently with requests of a different type.

Example 3.2. As shown in Figure 3.2, at time t_1, a read request of Type 1, $\mathcal{R}_1^{r1}$, is satisfied. The group of requests in Lane 3, $\mathcal{R}_2^{r3}$ and $\mathcal{R}_4^{r3}$, will be satisfied in the next phase, as indicated by the dark gray shading. In

Listing 1 R^{3}LP Definitions

```
type type_state: record
    in, out, head, sat, phase: unsigned integer initially 0
shared variables
    s: unsigned integer initially 0
    r1_type, r2_type, r3_type: type_state
constant
    R1_PRES       0x2  // Type 1 present bit
    R1_PHID       0x1  // Type 1 phase ID bit
    R1_BITS      0xff  // Type 1 bits in s
    R2_PRES     0x200  // Type 2 present bit
    R2_PHID     0x100  // Type 2 phase ID bit
    R2_BITS    0xff00  // Type 2 bits in s
    R3_PRES   0x20000  // Type 3 present bit
    R3_PHID   0x10000  // Type 3 phase ID bit
    R3_BITS 0xff0000  // Type 3 bits in s
```

the following phase, $\mathcal{R}_5^{r2}$ will be satisfied, as indicated with the light gray shading. As with the R^{2}LP, under the R^{3}LP, $\mathcal{R}_3^{r1}$ cannot be satisfied at t_1.

At time t_2, $\mathcal{R}_1^{r1}$ has completed and $\mathcal{R}_2^{r3}$ and $\mathcal{R}_4^{r3}$ are satisfied. $\mathcal{R}_5^{r2}$ will be satisfied in the next phase, and $\mathcal{R}_3^{r1}$ and the newly issued $\mathcal{R}_6^{r1}$ will be satisfied in the subsequent phase. At time t_3, $\mathcal{R}_2^{r3}$ and $\mathcal{R}_4^{r3}$ have completed, and $\mathcal{R}_5^{r2}$ is satisfied. $\Diamond$

This simple example gives intuition about how the R^{3}LP functions. Phases cycle between the three types, and the order of these phases depends on the order in which requests are issued and enqueued.

3.1.3 R^{3}LP Implementation

The following presents the pseudocode for an implementation of the R^{3}LP after first discussing the shared variables used within the R^{3}LP.

Shared variables of the R^{3}LP. The set of variables used by the R^{3}LP is presented in Listing 1. For each of the three types, a set of variables are defined as part of the *type_state*. The counters *in* and *out* represent how many requests of the specified type have been issued and have completed, respectively, similar to a ticket lock. The integer *head* indicates the ticket of the one request that modifies shared variables during the LOCK call and determines when all the requests in its phase are satisfied. The variable *sat* stores the highest satisfied ticket number, and the variable *phase* alternates between all 0's and all 1's to track different phases of this same type.

The variable *s*, as shown in Figure 3.3, is the shared variable on which the different request types synchronize. (The spacing between the pairs of bits for each phase is not required but makes the constant

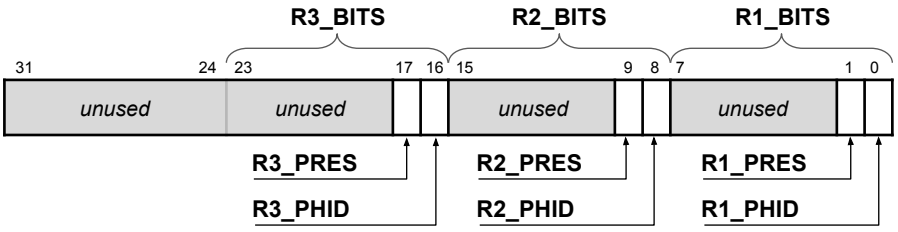


Figure 3.3: Bits in the shared *s* variable.

values more readable for this presentation.) In this implementation, *s* must be marked `volatile` to ensure stale values are never read, and operations on *s* are done via `__sync_*` functions to ensure atomic updates with necessary memory barriers.

Pseudocode for the R^{3}LP. The pseudocode for a request of Type 1 is shown in Listing 2. A request of Type 1, $\mathcal{R}_i^{r1}$, increments the *in* counter for Type 1, taking the previous value as its ticket value (Line 3). $\mathcal{R}_i^{r1}$ then checks if it is the "head" request (Line 4). If it is not, $\mathcal{R}_i^{r1}$ waits until its ticket number is at least the value *sat*, indicating that it is now satisfied (Lines 5-6),[3] or until it is the "head" request. The "head" request $\mathcal{R}_j^{r1}$ changes the phase (Line 7). Then, it sets the bits of *s* related to this specific type of request and queries the presence of requests of the other two types (Line 8). After separating both types (Lines 9-10), $\mathcal{R}_j^{r1}$ waits for a phase each of requests of Type 2 and 3, if necessary (Line 11). Specifically, for Type 2 it must ensure that there were no requests present ($r2 = 0$) or that those requests have completed ($r2 \neq (s\&\texttt{R2\_BITS})$). It performs the same checks for requests of Type 3. Finally, the request sets *sat* to indicate that access should be granted to all requests of Type 1 currently holding a ticket (the highest of which is ticket $in - 1$).

When a request $\mathcal{R}_i^{r1}$ completes, it increments *out* for its type (Line 16), then checks if it is the last request of the phase to complete (Line 17). If so, $\mathcal{R}_i^{r1}$ clears the bits of *s* that correspond to its type (Line 18) and sets the head to be the next ticket value (Line 19). This ticket may already be held by a request in the R^{3}LP_LOCK procedure.

The following lemma bounds the acquisition delay of the R^{3}LP based on the above description and implementation. In this lemma, L_{max}^{r1} (respectively, L_{max}^{r2} and L_{max}^{r3}) indicates the maximum critical-section length of a read request of Type 1 (respectively, Type 2 and Type 3).

[3]Line 5 can be modified to handle overflow in $p{\rightarrow}in$ and $p{\rightarrow}sat$. In a spin-based implementation, at most m requests can be active at once, so $p{\rightarrow}in$ and $p{\rightarrow}sat$ can be at most m apart. Therefore, the condition in Line 5 can be modified to $[p{\rightarrow}sat \geq ticket]$ **or** $[(p{\rightarrow}sat + m) \geq (ticket + m)]$ to mitigate overflow.

Listing 2 R^{3}LP Routine for Type 1

```
 1: procedure R3LP_LOCK(p: ptr to type_state)
 2:     var ticket,r2r3,r2,r3: unsigned int
 3:     ticket := fetch&add(p→in, 1)
 4:     while p→head ≠ ticket:                          ▷ Only head changes global variables
 5:         if p→sat ≥ ticket:
 6:             return                                  ▷ Satisfied
 7:     p→phase := ∼(p→phase)                           ▷ Flip phase bits
 8:     r2r3 := fetch&add(s, R1_PRES|(p→phase & R1_PHID))   ▷ Mark present
 9:     r2 := r2r3 & R2_BITS                            ▷ Get value for Type 2
10:     r3 := r2r3 & R3_BITS                            ▷ Get value for Type 3
11:     await (((r2 = 0) or (r2 ≠ (s & R2_BITS))) and ((r3 = 0) or (r3 ≠ (s & R3_BITS))))
12:     p→sat := p→in − 1                               ▷ Satisfied
13: end procedure

14: procedure R3LP_UNLOCK(p: ptr to type_state)
15:     var ticket: unsigned int
16:     ticket := fetch&add(p→out, 1)
17:     if p→sat = ticket:                              ▷ Last request of phase to finish
18:         fetch&and(s, ∼(R1_BITS))                    ▷ Clear R1_BITS
19:         p→head := ticket + 1                        ▷ Update head
20: end procedure
```

Lemma 3.1. *Under the R^{3}LP the worst-case acquisition delay of a request of any type is $L^{r1}_{max} + L^{r2}_{max} + L^{r3}_{max}$ time units.*

Proof. A request of a given type may need to wait for the completion of phases of each of the other two types of requests as well as a phase of its type. The implementation of the R^{3}LP in Listing 2 ensures that the duration of each such phase is at most the highest critical-section length of requests in that phase and that a phase of a given type is repeated after at most one phase of each of the other types.

Suppose we focus on a request of interest that is of Type 1. Suppose also that one or more requests of Type 1 are executing and that requests of both Type 2 and Type 3 are waiting at their corresponding Line 11. Our request of interest is not initially the head (Line 4) as one of the satisfied requests is. Any later issued requests of Type 1 also cannot become the head, so $p\rightarrow sat$ will not be updated and no new request of Type 1 can become satisfied. Thus, the current phase of requests of Type 1 will complete in at most L^{r1}_{max} time units, as that is the maximum critical-section length of any request of Type 1.

Once the satisfied requests of Type 1 complete, requests of either Type 2 or Type 3 may execute. Without loss of generality, suppose requests of Type 2 become satisfied. If our request of interest is not the head (Line 4), some other request of Type 1 is. The head request executes Lines 7-10 and waits at Line 11, as neither $r2$ nor $r3$ is zero and the phases represented in R2_BITS and R3_BITS have yet to change from the recorded values (taken in Lines 9 and 10). Once Line 8 is executed, any new requests of Type 2 will wait

at the corresponding Line 11 for Type 2 for the phase containing our request of interest to complete, as the phase shown in the R1_BITS of s will not change until after our request is satisfied and a request of Type 1 executes Line 18. Therefore, the phase of requests of Type 2 will complete in at most L^{r2}_{max} time units, by the definition of L^{r2}_{max}. Similarly all active requests of Type 3 will become satisfied, but any new requests must wait, and the phase of requests of Type 3 will complete within L^{r3}_{max} time units. Thus, our request of interest of Type 1 may experience acquisition delay of up to $L^{r1}_{max} + L^{r2}_{max} + L^{r3}_{max}$ time units in the worst case.

The analysis above applies to requests of Type 2 and Type 3 as well. If any of the types do not have an active request while a request is active, it can only shorten the blocking experienced. Therefore, the worst-case acquisition delay of a request of any type is $L^{r1}_{max} + L^{r2}_{max} + L^{r3}_{max}$ time units under the R^{3}LP. □

Corollary 3.1. *The bound given in Lemma 3.1 is tight.*

Proof. This is illustrated in Example 3.2 and Figure 3.2 with $\mathcal{R}^{r1}_3$. □

Corollary 3.2. *Under the R^{3}LP, if no requests of Type 1 are present, the worst-case acquisition delay of a request of Type 2 or Type 3 is $L^{r2}_{max} + L^{r3}_{max}$ time units.*

Proof. This follows directly from the proof of Lemma 3.1, as one fewer phase of execution is possible before a request becomes satisfied. □

The above proof is actually phase independent, so similar bounds exist regardless of which of the types of request is not present.

3.2 The Fast RW-RNLP

The fast RW-RNLP is constructed in a modular fashion based on existing locking protocols and a choice of two new protocols. These new protocols are the R^{3}LP and the RW-RNLP*, which is a new variant of the RW-RNLP. In this section, the structure of the fast RW-RNLP is presented and abstract pi-blocking analysis is provided. Next, the pi-blocking analysis for the fast RW-RNLP variant with the R^{3}LP is presented. Finally, the RW-RNLP* is described and the pi-blocking analysis for the fast RW-RNLP variant with the RW-RNLP* is presented.

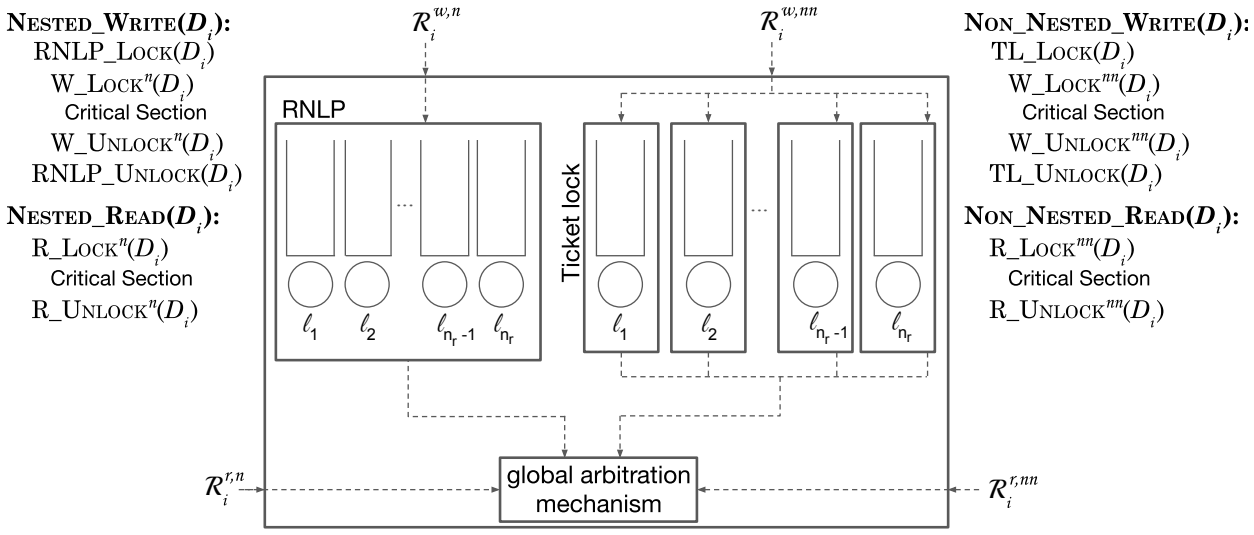


Figure 3.4: Fast RW-RNLP structure.

3.2.1 Protocol Structure

This section presents a description of the fast RW-RNLP protocol. The goals for this protocol are threefold: **(i)** read and non-nested write requests should have low lock/unlock overhead; **(ii)** such requests should have contention-sensitive worst-case pi-blocking bounds; **(iii)** nested write requests should have worst-case pi-blocking bounds that are asymptotically the same as under the RW-RNLP. To achieve Goals (ii) and (iii), requests are separated by type; Section 3.2.2 and Section 3.2.5 show that these goals can be achieved with the R^{3}LP or the RW-RNLP*, respectively. (Between these two protocols, there is a tradeoff between optimizing for better analytical bounds and optimizing for better runtime performance.) Goal (i) is addressed in Section 3.3 with an experimental evaluation of both implementations.

The fast RW-RNLP is defined by using the lock and unlock routines of other locking protocols as subroutines. As shown in Figure 3.4, ordinary (not phase-fair) mutex ticket locks (TLs) (Mellor-Crummey and Scott, 1991a) and the RNLP (Ward and Anderson, 2012) are used.

A non-nested write request first acquires a TL associated with its requested resource. A FIFO-ordered TL provides mutex sharing for a single resource and ensures contention-sensitive pi-blocking. The lemma below follows from the definition of a ticket lock.

Lemma 3.2. *The worst-case acquisition delay of a request $\mathcal{R}_i$ under a ticket lock is upper bounded by the product of the number of requests ahead of $\mathcal{R}_i$ and the longest time any such request holds the lock.*

Similarly, a nested write request invokes the RNLP; recall from Section 2.6.3.5 that the RNLP provides mutex sharing and supports nested requests. Under it, the worst-case pi-blocking of any request is $O(m)$ (Ward and Anderson, 2012). More specifically, the following lemma bounds acquisition delay in this context.

Lemma 3.3. *(Ward and Anderson, 2012) The worst-case acquisition delay of a request $\mathcal{R}_i$ under the RNLP is upper bounded by the product of the number of previously issued active requests in the system and the longest time any such request holds the lock.*

Note that instead of the RNLP, a different protocol could be used to arbitrate between nested write requests without changing the overall structure of the fast RW-RNLP. Unless indicated otherwise, assume the RNLP is used.

To arbitrate between the two types of write requests, as well as read requests, a global arbitration mechanism is needed. Once a non-nested (respectively, nested) write request is granted the resource-specific lock by a TL (respectively, the RNLP), it may enter the global arbitration mechanism, as depicted in Figure 3.4. Any read requests may enter the global arbitration mechanism directly.

The global arbitration mechanism is applied in a specific context in the fast RW-RNLP, which can be described by the following rules. With the exception of Rule P3, the rules below are standard for non-preemptive spin-based locking protocols. As we shall see, Rule P3 enforces the restricted context and enables contention-sensitive pi-blocking bounds for non-nested requests to be computed in the context of the fast RW-RNLP. It is upheld by the structure explained above and depicted in Figure 3.4. Rule P3 is also trivially upheld in systems with only single-writer resources.

P1 A resource-holding job is always scheduled.

P2 At most m jobs may have incomplete resource requests at any time, at most one per processor.

P3 There is at most one incomplete non-nested write request and one incomplete nested write request per resource at any time.

In this chapter, two ways of implementing the global arbitration mechanism are presented: the R^{3}LP, presented earlier in Section 3.1, and a restricted variant of the RW-RNLP, the RW-RNLP*, presented later in Section 3.2.3. Figure 3.4 shows how each type of request uses these locking protocols.[4]

[4]The lock and unlock routines for the R^{3}LP or RW-RNLP* routines have been denoted in a slightly abbreviated way. For example, W_LOCK^{nn} denotes the lock routine invoked by non-nested write requests under the chosen protocol.

The worst-case acquisition delay of a read request under the global arbitration mechanism is denoted G^r. Similarly, the worst-case acquisition delay of a non-nested write request (respectively, nested write request) under the global arbitration mechanism is denoted $G^{w,nn}$ (respectively, $G^{w,n}$). The following theorem incorporates these upper bounds directly. The maximum critical-section length of each request type is defined similarly; for example, $L^{w,nn}_{max}$ is the maximum critical-section length of any non-nested write request.

Theorem 3.1. *Under the fast RW-RNLP, the worst-case acquisition delay of a request $\mathcal{R}_i$ is:*

(i) *G^r time units, if $\mathcal{R}^r_i$ is a read request;*

(ii) *$N_i \cdot (L^{w,nn}_{max} + G^{w,nn}) + G^{w,nn}$ time units, if $\mathcal{R}^{w,nn}_i$ is a non-nested write request;*

(iii) *$(m-1) \cdot (L^{w,n}_{max} + G^{w,n}) + G^{w,n}$ time units, if $\mathcal{R}^{w,n}_i$ is a nested write request.*

Proof. In Case (i), a read request $\mathcal{R}^r_i$ enters the global arbitration protocol directly. Therefore, the worst-case acquisition delay of $\mathcal{R}^r_i$ is G^r time units.

In Case (ii), a request $\mathcal{R}^{w,nn}_i$ must wait for each contending write request ahead of it in the TL associated with its requested resource. There may be up to N_i contending write requests, each of which may face an acquisition delay of up to $G^{w,nn}$ time units within the global arbitration protocol while holding the ticket lock. Additionally, each such request must then execute its critical section for up to $L^{w,nn}_{max}$ time units. Thus, $\mathcal{R}^{w,nn}_i$ may wait up to $N_i \cdot (L^{w,nn}_{max} + G^{w,nn})$ time units before invoking the global arbitration protocol (Lemma 3.2), after which it may experience an acquisition delay of up to $G^{w,nn}$ time units. This yields a worst-case acquisition delay of $N_i \cdot (L^{w,nn}_{max} + G^{w,nn}) + G^{w,nn}$ time units for $\mathcal{R}^{w,nn}_i$.

A request $\mathcal{R}^{w,n}_i$ in Case (iii) must wait for other requests within the RNLP to complete before invoking the global arbitration protocol. There may be up to $m-1$ such requests (Lemma 3.3 and Rule P2). By using the same argument as before and applying Lemma 3.3, the worst-case acquisition delay of $\mathcal{R}^{w,n}$ is $(m-1) \cdot (L^{w,n}_{max} + G^{w,n}) + G^{w,n}$ time units. □

3.2.2 The Fast RW-RNLP with the R^{3}LP

This section describes how to apply the R^{3}LP as the global arbitration mechanism based on the structure of the fast RW-RNLP provided in the previous section. Then, the worst-case acquisition delay a request of each type may experience under the fast RW-RNLP with the R^{3}LP is examined to show that this variant of the fast RW-RNLP achieves Goals (ii) and (iii) presented in Section 3.2.1.

Given that Rule P3 ensures that there is at most one non-nested write request submitted to the R^{3}LP per resource at any given time, all non-nested write requests that are present can be allowed to execute together; they must require different resources. In a sense, this means that all non-nested write requests can be treated similarly to read requests relative to each other. The same holds true for the set of nested write requests at a given time. Naturally, all read requests, nested or non-nested, can execute together.

It follows from this discussion that, to coordinate nested and non-nested write requests as well as read requests, it suffices to use a protocol that can coordinate three different types of read requests: requests of the same type can execute concurrently but requests of different types cannot. The R^{3}LP can be used for this purpose. The three types of requests processed by this application of the R^{3}LP are non-nested write, nested write, and read requests.

Next, the worst-case acquisition delay that any request can experience under the fast RW-RNLP with the R^{3}LP is bounded. These requests are distinguished by type: an arbitrary read request $\mathcal{R}_i^r$, non-nested write request $\mathcal{R}_i^{w,nn}$, and nested write request $\mathcal{R}_i^{w,n}$.

Theorem 3.2. *Under the fast RW-RNLP with the R^{3}LP, the worst-case acquisition delay for a request $\mathcal{R}_i$ is:*
(i) $L_{max}^w + L_{max}^r$ *time units, if $\mathcal{R}_i^r$ is a read request and no nested write requests are active while $\mathcal{R}_i^r$ is active;*
(ii) $2L_{max}^w + L_{max}^r$ *time units, if $\mathcal{R}_i^r$ is a read request and nested write requests may be active while $\mathcal{R}_i^r$ is active;*
(iii) $N_i \cdot (2L_{max}^w + L_{max}^r) + L_{max}^w + L_{max}^r$ *time units, if $\mathcal{R}_i^{w,nn}$ is a non-nested write request and no nested write requests are active while $\mathcal{R}_i^{w,nn}$ is active;*
(iv) $N_i \cdot (3L_{max}^w + L_{max}^r) + 2L_{max}^w + L_{max}^r$ *time units, if $\mathcal{R}_i^{w,nn}$ is a non-nested write request and nested write requests may be active while $\mathcal{R}_i^{w,nn}$ is active;*
(v) $(m-1) \cdot (3L_{max}^w + L_{max}^r) + 2L_{max}^w + L_{max}^r$ *time units, if $\mathcal{R}_i^{w,n}$ is a nested write request.*

Proof. Cases (ii), (iv), and (v) follow directly from Lemma 3.1 and Theorem 3.1. Here, $G^r = G^{w,nn} = G^{w,n} = 2L_{max}^w + L_{max}^r$.

When no nested write requests are active, as in Cases (i) and (iii), the above statements follow from Corollary 3.2 and Theorem 3.1. Here, $G^r = G^{w,nn} = L_{max}^w + L_{max}^r$; there is no nested write phase. □

Theorem 3.2 shows that non-nested requests have contention-sensitive blocking (Goal (ii)) and that nested requests have blocking bounds asymptotically the same as under the RW-RNLP (Goal (iii)). Referring to Goal (iii), note that the worst-case pi-blocking under the RW-RNLP is $O(1)$ for read requests and $O(m)$

for write requests (Ward and Anderson, 2014).[5] The pi-blocking bound for nested write requests under the fast RW-RNLP with the R^{3}LP has a higher coefficient than under the RW-RNLP. In practice, however, the fast RW-RNLP with the R^{3}LP outperforms the RW-RNLP, as discussed in Section 3.3.

3.2.3 The RW-RNLP*

While the R^{3}LP arbitrates resource access correctly, it does so at the cost of some concurrency; access to resources is coordinated in global phases. An alternate choice for the global arbitration mechanism is the RW-RNLP*. Similarly to the R^{3}LP, the RW-RNLP* must arbitrate resource access between read and write requests. However, it arbitrates access on a per-resource basis, allowing increased concurrency in resource accesses. It is obtained from the RW-RNLP by altering one aspect of its design and changing the context in which it is applied (Rule P3). For each resource ℓ_a, the RW-RNLP* maintains two queues Q_a^r and Q_a^w, for unsatisfied read and write requests, respectively.

Example 3.3. Figure 3.5 is used as a continuing example to illustrate important concepts in the design of the RW-RNLP*. Each inset of this figure shows read and write queues for four resources: ℓ_1, ℓ_2, ℓ_3, and ℓ_4. At the time illustrated in Figure 3.5 (a), the write request $\mathcal{R}_1^w$ is satisfied for its requested resources $D_1 = \{\ell_1, \ell_2\}$, as indicated by being positioned within the circles denoting the resources ℓ_1 and ℓ_2. Because $\mathcal{R}_1^w$ is satisfied, it is not in any of the queues. Similarly, the read request $\mathcal{R}_2^r$ for $D_2 = \{\ell_3, \ell_4\}$ is satisfied. $\Diamond$

Basic RW-RNLP* rules. The RW-RNLP* is described via a set of rules to which an implementation must conform. The following are general rules that define how requests are processed.

G1 When J_i issues $\mathcal{R}_i$ at time t, the timestamp of the request is recorded: $ts(\mathcal{R}_i) := t$.

G2 When $\mathcal{R}_i$ is satisfied, it is dequeued from either Q_a^r (if it is a read request) or Q_a^w (if it is a write request) for each $\ell_a \in D_i$.

G3 When $\mathcal{R}_i$ completes, it unlocks all resources in D_i.

G4 Each request issuance or completion occurs atomically. Therefore, there is a total order on timestamps, and a request cannot be issued at the same time that a critical section completes.

[5] More precisely, the bounds presented are $L_{max}^w + L_{max}^r$ and $(m-1)(L_{max}^w + L_{max}^r)$ for read and write requests, respectively.

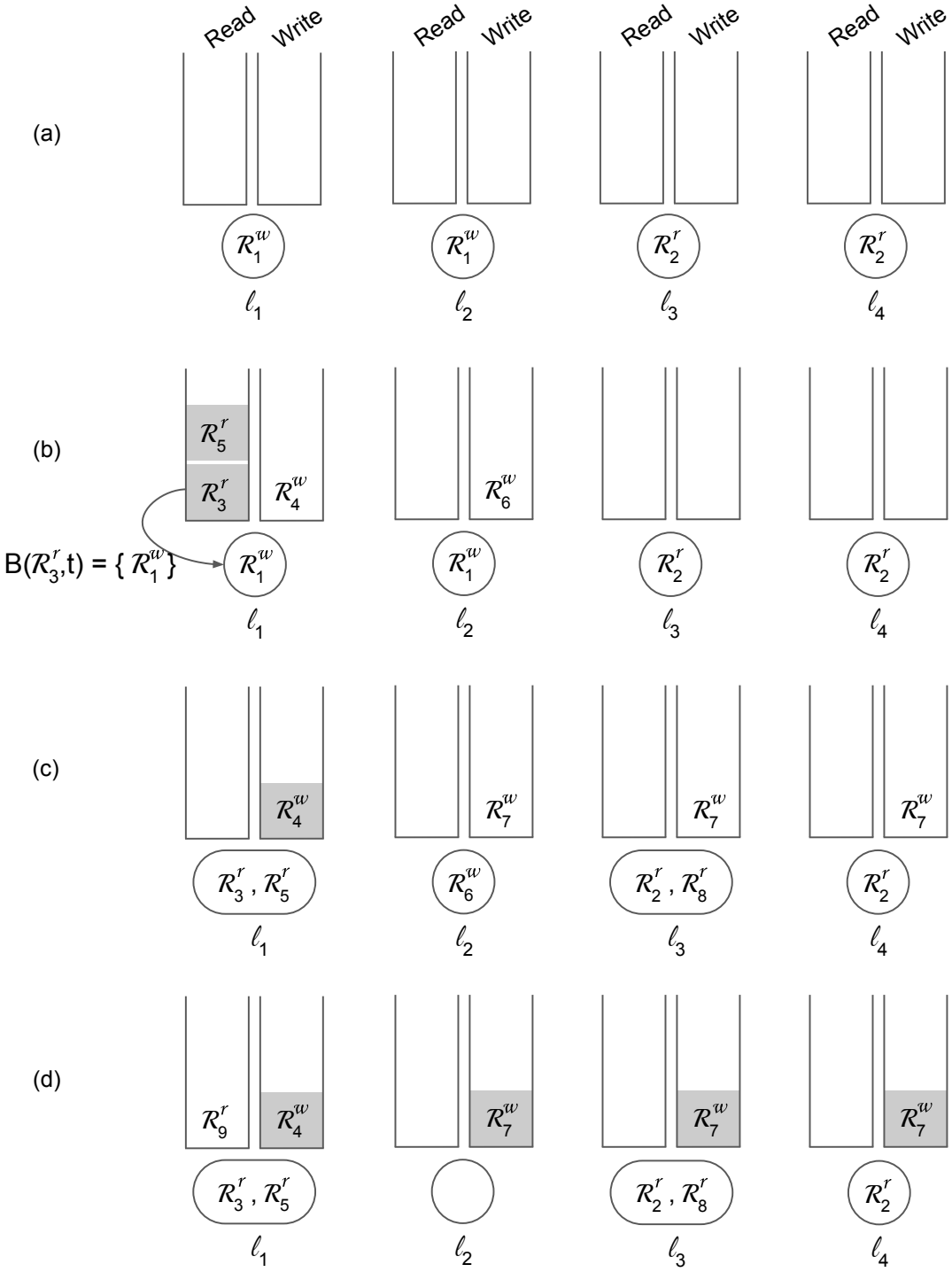


Figure 3.5: Example illustrating the rules of the RW-RNLP*.

Example 3.3 (continued). Moving from inset (a) to inset (b) in Figure 3.5, four additional requests have been issued. Timestamps are determined for these requests when they are issued (Rule G1). The issuance of each request occurs atomically (Rule G4), so it is not possible for two requests to obtain the same timestamp.

The arrow from $\mathcal{R}_3^r$ to $\mathcal{R}_1^w$ indicates that $\mathcal{R}_3^r$ is blocked by $\mathcal{R}_1^w$. This blocking relationship is formally defined later and serves to represent just one such relationship in the system.

Figure 3.5 (c) depicts the system after $\mathcal{R}_1^w$ has completed. By Rule G3, it released resources ℓ_1 and ℓ_2. This enabled both $\mathcal{R}_3^r$ and $\mathcal{R}_5^r$ to be satisfied for ℓ_1 and dequeued from Q_1^r (Rule G2). Similarly, $\mathcal{R}_6^w$ became satisfied for ℓ_2.

In moving from inset (b) to inset (c), $\mathcal{R}_7^w$ and $\mathcal{R}_8^r$ have been issued, and $\mathcal{R}_8^r$ was satisfied immediately. Notice that request $\mathcal{R}_7^w$ for resources $D_7 = \{\ell_2, \ell_3, \ell_4\}$ was atomically enqueued on Q_2^w, Q_3^w, and Q_4^w. Because such an action is atomic, no cycles among blocked requests can exist. In an actual implementation, the issuance and completion of a request would not really occur atomically. However, an implementation must ensure that these actions have the "effect" of being atomic. Section 3.2.6 considers such issues. ◊

Read and write entitlement. Like the RW-RNLP, the RW-RNLP* functions by alternating read and write phases. The mechanism for orchestrating these phases is *entitlement*, which is defined separately for read and write requests below (these definitions are taken directly from (Ward and Anderson, 2014)). Intuitively, a request is entitled when it should be satisfied in the next phase, thus only *unsatisfied* requests may be entitled. Together with the reader and writer rules presented later, the definition of entitlement ensures progress and allows us to upper-bound pi-blocking times. Below, $E(Q_a^w)$ is used to denote the earliest-timestamped unsatisfied write request for resource ℓ_a.

Example 3.3 (continued). In Figure 3.5 (b), $E(Q_2^w) = \mathcal{R}_6^w$. ◊

Definition 3.1. An unsatisfied read request $\mathcal{R}_i^r$ becomes *entitled* when there exists $\ell_a \in D_i$ that is write locked, and for each resource $\ell_a \in D_i$, $E(Q_a^w)$ is not entitled (see Definition 3.2).[6] (Note that $E(Q_a^w) = \emptyset$ could hold. In this case, $E(Q_a^w) = \emptyset$ is considered to be a "null" request that is not entitled.) $\mathcal{R}_i^r$ remains entitled until it is satisfied.

[6]Entitlement is a property of a request, and Definition 3.1 and Definition 3.2 give conditions upon which a request becomes entitled in terms of the entitlement of other requests. Therefore, while Definition 3.1 and Definition 3.2 reference each other parenthetically to aid the reader, they are not in fact circularly defined.

Definition 3.2. An unsatisfied write request $\mathcal{R}_i^w$ becomes *entitled* when for each $\ell_a \in D_i$, $\mathcal{R}_i^w = E(Q_a^w)$, no read request in Q_a^r is entitled (see Definition 3.1),[6] and ℓ_a is not write locked. $\mathcal{R}_i^w$ remains entitled until it is satisfied.

Example 3.3 (continued). In Figure 3.5 (b), $\mathcal{R}_3^r$ and $\mathcal{R}_5^r$ are both entitled (Definition 3.1): ℓ_1 is write locked, and there exists no resource ℓ_a in D_3 or D_5 for which $E(Q_a^w)$ is entitled (Definition 3.2). Entitled requests are indicated in Figure 3.5 by gray shading. In Figure 3.5 (c), $\mathcal{R}_4^w$ is entitled: ℓ_1 is the only resource in D_4, $E(Q_1^w) = \mathcal{R}_4^w$ holds, there is no entitled read in Q_1^r, and ℓ_1 is not write locked. In moving from inset (c) to inset (d), $\mathcal{R}_6^w$ completed and released ℓ_2. In Figure 3.5 (d), $\mathcal{R}_7^w$ is entitled: $\mathcal{R}_7^w$ was at the head of each of its queues and there were no entitled read requests in the corresponding read queues, so the only condition that prevented $\mathcal{R}_7^w$ from being entitled earlier was $\mathcal{R}_6^w$'s lock on ℓ_2. ◊

Rules for read and write requests. The specification of the RW-RNLP* is completed by stating rules that govern how read and write requests are processed. To state these rules, notation is required to identify the set of requests on which an entitled request $\mathcal{R}_i$ (a read or a write) is blocked. Specifically, $B(\mathcal{R}_i, t)$ denotes the set of requests on which such a request $\mathcal{R}_i$ is blocked at time t.

Example 3.3 (continued). In Figure 3.5 (b), there are two entitled requests, $\mathcal{R}_3^r$ and $\mathcal{R}_5^r$, both waiting on the satisfied write request $\mathcal{R}_1^w$. If inset (b) reflects the system state at time t, then $B(\mathcal{R}_3^r, t) = \{\mathcal{R}_1^w\}$ and $B(\mathcal{R}_5^r, t) = \{\mathcal{R}_1^w\}$. Only one of these relationships is depicted with an arrow in the diagram to avoid clutter. Similarly, if Figure 3.5 (c) reflects the system state at time t', then $B(\mathcal{R}_4^w, t') = \{\mathcal{R}_3^r, \mathcal{R}_5^r\}$. Note that there are other blocking relationships throughout Figure 3.5, and $B(\mathcal{R}_i, t)$ is only defined for $\mathcal{R}_i$ at a time t when $\mathcal{R}_i$ is entitled. ◊

The rules for read requests are as follows.

R1 When $\mathcal{R}_i^r$ is issued, for each $\ell_a \in D_i$, $\mathcal{R}_i^r$ is enqueued in Q_a^r. If $\mathcal{R}_i^r$ does not conflict with any entitled or satisfied write requests, then it is satisfied immediately.

R2 An entitled read request $\mathcal{R}_i^r$ is satisfied at the first time instant t such that $B(\mathcal{R}_i^r, t) = \emptyset$.

Example 3.3 (continued). When $\mathcal{R}_3^r$ and $\mathcal{R}_5^r$ were issued, by Rule R1, each was enqueued in Q_1^r, as shown in Figure 3.5 (b). When $\mathcal{R}_1^w$ later completed at some time t, as shown in Figure 3.5 (c), $B(\mathcal{R}_3^r, t) = \emptyset$ and $B(\mathcal{R}_5^r, t) = \emptyset$ were both established and $\mathcal{R}_3^r$ and $\mathcal{R}_5^r$ were both satisfied immediately, by Rule R2. Figure 3.5 (c)

also shows $\mathcal{R}_8^r$ being satisfied immediately after being issued. This occurred by Rule R1, as no satisfied or entitled write requests for ℓ_3 existed at that time. $\Diamond$

The rules for write requests are as follows.

W1 When $\mathcal{R}_i^w$ is issued, for each $\ell_a \in D_i$, $\mathcal{R}_i^w$ is enqueued in timestamp order in the write queue Q_a^w. If $\mathcal{R}_i^w$ does not conflict with any entitled or satisfied requests (read or write), then it is satisfied immediately.

W2 An entitled write request $\mathcal{R}_i^w$ is satisfied at the first time instant t such that $B(\mathcal{R}_i^w, t) = \emptyset$.

Example 3.3 (continued). When $\mathcal{R}_6^w$ was issued prior to the system state depicted in Figure 3.5 (b), it was enqueued in Q_2^w, and because it conflicted with the satisfied request $\mathcal{R}_1^w$, by Rule W1, it was not satisfied immediately. Request $\mathcal{R}_1^w$ later completed at some time t, as shown in Figure 3.5 (c), and at that time t, $\mathcal{R}_6^w$ became entitled and $B(\mathcal{R}_6^w, t) = \emptyset$ held, so $\mathcal{R}_6^w$ became satisfied, by Rule W2. $\Diamond$

Write expansion. Aside from Rule P3, the only other difference between the RW-RNLP* and the RW-RNLP is with regard to a technique called *write expansion*, which is employed by the latter but not the former. Since the RW-RNLP* does not employ write expansion, the necessary formal machinery to completely define this technique is not introduced here. Instead, the general idea behind write expansion is conveyed with an example.

Example 3.4. The general idea behind write expansion is as follows. If a write request $\mathcal{R}_i^w$ is issued, and if a read request $\mathcal{R}_j^r$ that accesses resources in common with $\mathcal{R}_i^w$ could *possibly* be active concurrently, then the set of resources requested by $\mathcal{R}_i^w$, D_i, must be expanded to include all resources in D_j. An example is given in Figure 3.6. In inset (a), a write request $\mathcal{R}_1^w$ is satisfied, holding the lock for ℓ_3. Inset (b) shows two possible scenarios after the issuance of $\mathcal{R}_2^w$, $\mathcal{R}_3^w$, and $\mathcal{R}_4^r$, with $D_2 = \{\ell_2, \ell_3\}$, $D_3 = \{\ell_1\}$, and $D_4 = \{\ell_1, \ell_2\}$. Inset (b)(i), on the left, shows the situation with no write expansion. $\mathcal{R}_3^w$ requires only resource ℓ_1 and thus is immediately satisfied. $\mathcal{R}_4^r$ is then entitled. In inset (b)(ii), $\mathcal{R}_2^w$ and $\mathcal{R}_3^w$ are expanded: because there exists a read request (namely, $\mathcal{R}_4^r$) in the system that requires ℓ_1 and ℓ_2, $\mathcal{R}_2^w$ must be issued for $D_2 = \{\ell_1, \ell_2, \ell_3\}$ and $\mathcal{R}_3^w$ must be issued for $D_3 = \{\ell_1, \ell_2\}$. Therefore, in inset (b)(ii), $\mathcal{R}_3^w$ cannot be satisfied until $\mathcal{R}_2^w$ completes, even though they do not share resources.

Inset (c) shows the situation after $\mathcal{R}_1^w$ has completed. As depicted in inset (c)(i), in the scenario without write expansion, nothing new happens to the other requests, as $\mathcal{R}_2^w$ cannot proceed ahead of the entitled read $\mathcal{R}_4^r$. However, as shown in inset (c)(ii), in the scenario with write expansion, the completion of $\mathcal{R}_1^w$ makes $\mathcal{R}_2^w$ entitled. $\Diamond$

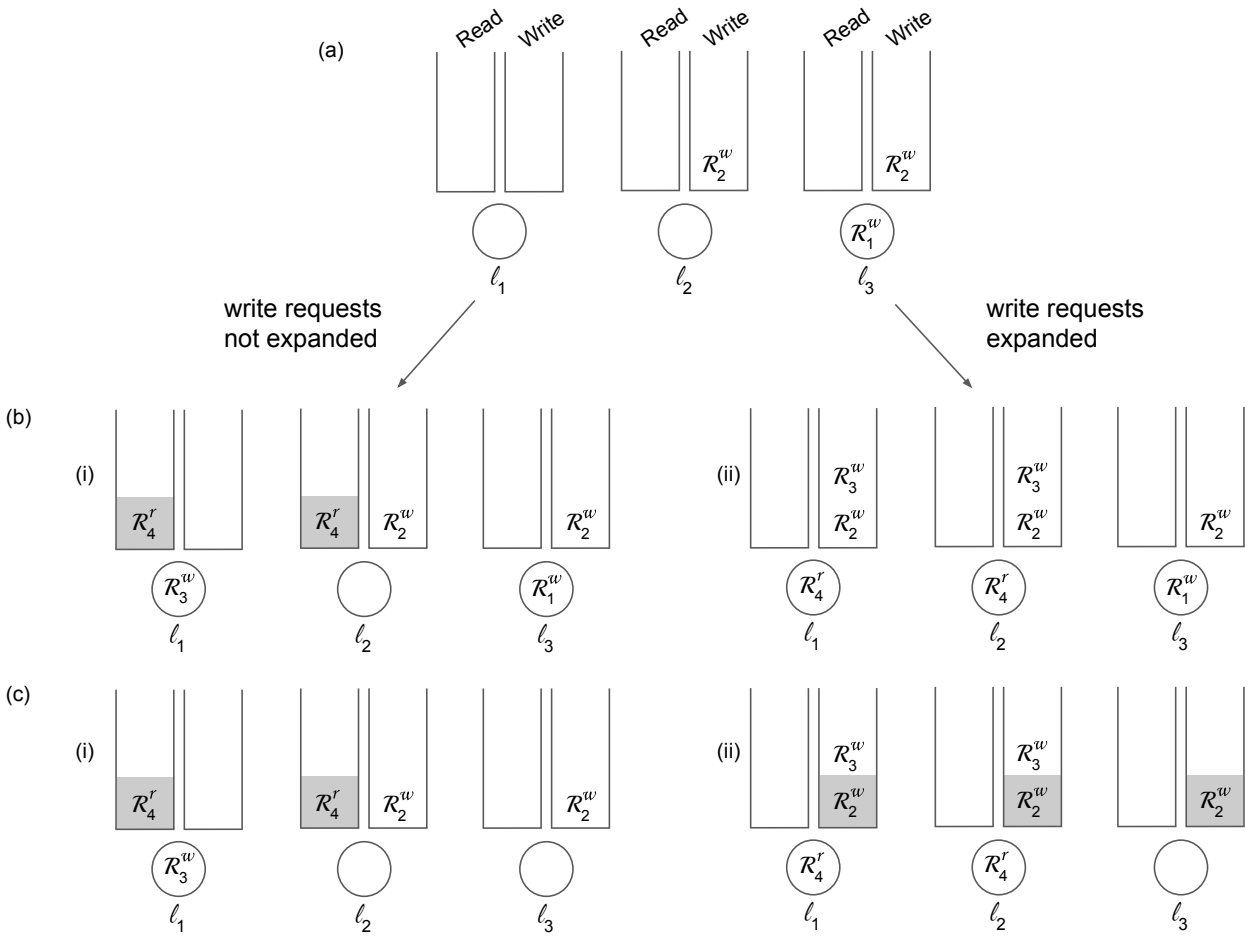


Figure 3.6: System states without write expansion are labeled **(i)**, and states with write expansion (used in the RW-RNLP) are labeled **(ii).**

One reason write expansion is used in the RW-RNLP is because it makes reasoning about the largest possible pi-blocking for write requests easier. With write expansion, if $\mathcal{R}_i^w$ is the earliest-timestamped write among *all* write requests, then it is either entitled or satisfied, as illustrated in Example 3.4 and proven by Ward and Anderson (2014). Additionally, write expansion eases certain implementation challenges.

In the context of the global arbitration mechanism, write expansion is problematic, as the ultimate intent is to speed the processing of non-nested requests. With write expansion, these could be converted into nested requests. However, removing write expansion under the RW-RNLP* creates additional complexity with respect to the pi-blocking scenarios that can occur, and increases worst-case pi-blocking bounds for write requests by a constant factor compared to the bounds under the RW-RNLP.

3.2.4 RW-RNLP* Pi-Blocking Bounds

This section presents bounds on the worst-case acquisition delay experienced by a request under the RW-RNLP*. The properties needed to derive acquisition-delay bounds are stated below. Lemma 3.4 and Theorem 3.3 were proven by Ward and Anderson (2014) (appearing as Lemma 1 and Theorem 1 in that paper), and those proofs are not affected by the changes made to the RW-RNLP to obtain the RW-RNLP*. The remaining properties either require new proofs or are entirely new. Each of these properties is illustrated below by refering to the prior example.

Lemma 3.4. *Under the RW-RNLP*, a write request $\mathcal{R}_i^w$ experiences acquisition delay of at most L_{max}^r time units after becoming entitled.*

Example 3.3 (continued). In insets (c) and (d) of Figure 3.5, $\mathcal{R}_4^w$ is simply waiting for all requests in $B(\mathcal{R}_4^w, t_e)$ to complete, where t_e is the time when $\mathcal{R}_4^w$ became entitled. It can be shown that no new requests can be added to $B(\mathcal{R}_4^w, t_e)$ until $\mathcal{R}_4^w$ is satisfied. Furthermore, by Definition 3.2, all of the requests in this set are read requests. In this scenario, $\mathcal{R}_4^w$ waits for two requests to complete before becoming satisfied, as $B(\mathcal{R}_4^w, t_e) = \{\mathcal{R}_3^r, \mathcal{R}_5^r\}$. In the worst case, $\mathcal{R}_4^w$ must wait for L_{max}^r time units. Note that having multiple reads in the set $B(\mathcal{R}_4^w, t_e)$ does not increase this worst-case acquisition delay. ◊

Theorem 3.3. *Under the RW-RNLP*, the worst-case acquisition delay of a read request $\mathcal{R}_i^r$ is at most $L_{max}^w + L_{max}^r$ time units.*

Example 3.3 (continued). Consider $\mathcal{R}_9^r$ in Figure 3.5 (d). Resource ℓ_1 is currently in a read phase, as $\mathcal{R}_3^r$ and $\mathcal{R}_5^r$ are in their critical sections, and there is an entitled write request, $\mathcal{R}_4^w$. Therefore, before $\mathcal{R}_9^r$ is satisfied, the read requests $\mathcal{R}_3^r$ and $\mathcal{R}_5^r$ could take up to L_{max}^r time units, and then the write request $\mathcal{R}_4^w$ could take up to L_{max}^w additional time units. ◊

Lemma 3.5 below is very similar to Lemma 2 in the paper that presented the RW-RNLP*, and much of the proof given for it is taken verbatim from there (Ward and Anderson, 2014). However, new reasoning is required as write expansion is not employed here.

Lemma 3.5. *Under the RW-RNLP*, if $\mathcal{R}_i^w$ is the earliest-timestamped active write request for each resource in D_i, then $\mathcal{R}_i^w$ will be satisfied within $L_{max}^w + L_{max}^r$ time units.*

Proof. An unsatisfied write request $\mathcal{R}_i^w$ is either entitled or not. If $\mathcal{R}_i^w$ is entitled, then by Lemma 3.4, it will become satisfied within L_{max}^r time units. Otherwise, by Definition 3.2, for some resource $\ell_a \in D_i$, either **(i)** $\mathcal{R}_i^w \neq E(Q_a^w)$, **(ii)** some request $\mathcal{R}_x^r \in Q_a^r$ is entitled, or **(iii)** ℓ_a is write locked by some other request. By Rule W1, Cases (i) and (iii) are not possible because the write queues are timestamp ordered, and $\mathcal{R}_i^w$ is the earliest-timestamped active write request for each resource in D_i. For Case (ii), assume that $\mathcal{R}_x^r$ is entitled and $\ell_a \in D_i \cap D_x$. Then, by Definition 3.1, $\mathcal{R}_x^r$ is blocked by at least one satisfied write request $\mathcal{R}_j^w$. By Rule P1 (a resource-holding job is continually scheduled), all such write requests will complete within L_{max}^w time units. At the time t when all such write requests have completed, by Rule R2, each $\mathcal{R}_x^r$ in $B(\mathcal{R}_i^w, t)$ will be satisfied, and by Definition 3.2, $\mathcal{R}_i^w$ will be entitled. By Lemma 3.4, $\mathcal{R}_i^w$ will subsequently experience at most L_{max}^r additional time units of delay before being satisfied. □

In systems for which each resource is a single-writer resource, each write request is the earliest-timestamped active write request for all of its required resources upon issuance.

Corollary 3.3. *Under the RW-RNLP*, if all resources are single-writer resources, then the worst-case acquisition delay of a write request $\mathcal{R}_i^w$ is at most $L_{max}^w + L_{max}^r$ time units.*

The time it takes the earliest-timestamped active write request to become satisfied in the special scenario of no active nested requests is similarly bounded.

Lemma 3.6. *Under the RW-RNLP*, if no nested requests are active while the non-nested request $\mathcal{R}_i^{w,nn}$ is active, and if $\mathcal{R}_i^{w,nn}$ is the earliest-timestamped active write request for its lone requested resource ℓ_a in D_i, then $\mathcal{R}_i^{w,nn}$ will be satisfied within L_{max}^r time units.*

Proof. The proof of this lemma differs from that given above for Lemma 3.5 only in how Case (ii) is addressed. For Case (ii) in the context of Lemma 3.6, if the non-nested request $\mathcal{R}_x^{r,nn}$ is entitled, then by Definition 3.1, it must blocked by a satisfied write request $\mathcal{R}_j^{w,nn}$ for resource ℓ_a. However, $\mathcal{R}_i^{w,nn}$ is the earliest-timestamped request for ℓ_a, so Case (ii) is actually impossible in the context of Lemma 3.6. Therefore, $\mathcal{R}_i^{w,nn}$ must be either satisfied or entitled, and in the latter case, it becomes satisfied within L_{max}^r time units, by Lemma 3.4. □

The next two lemmas heavily exploit Rule P3, which requires that there be at most one incomplete non-nested write request and one incomplete nested write request per resource at any time.

Lemma 3.7. *Under the RW-RNLP*, after being issued, a nested write request $\mathcal{R}_i^{w,n}$ will become the earliest-timestamped active write request for all of the resources in D_i within $2L_{max}^w + L_{max}^r$ time units.*

Proof. For any resource in D_i for which $\mathcal{R}_i^{w,n}$ is not the earliest-timestamped write request, by Rule P3, the earliest-timestamped write is a non-nested write request. By Lemma 3.5, each such request is satisfied within $L_{max}^w + L_{max}^r$ time units. By Rule P1, once satisfied, all such non-nested write requests will complete within L_{max}^w time units. Summing these two bounds yields the worst-case bound of $2L_{max}^w + L_{max}^r$ time units. □

Lemma 3.8. *Under the RW-RNLP*, after being issued, a non-nested write request $\mathcal{R}_i^{w,nn}$ will become the earliest-timestamped active write request for its lone requested resource ℓ_a in D_i:*
(i) *immediately, if no nested write requests are active while $\mathcal{R}_i^{w,nn}$ is active;*
(ii) *within $4L_{max}^w + 2L_{max}^r$ time units, if nested requests may be active while $\mathcal{R}_i^{w,nn}$ is active.*

Proof. In Case (i), by Rule P3, there are no other write requests accessing ℓ_a, so $\mathcal{R}_i^{w,nn}$ immediately becomes the earliest-timestamped request for that resource.

In Case (ii), if $\mathcal{R}_i^{w,nn}$ is not immediately the earliest-timestamped write request for ℓ_a, then there exists exactly one nested write request $\mathcal{R}_x^{w,n}$ that is the earliest-timestamped write request for ℓ_a (Rule P3). By Lemma 3.7, $\mathcal{R}_x^{w,n}$ will be the earliest-timestamped request for *all* of its requested resources within $2L_{max}^w + L_{max}^r$ time units. By Lemma 3.5, $\mathcal{R}_x^{w,n}$ will be satisfied within an additional $L_{max}^w + L_{max}^r$ time units. Once it is satisfied, by Rule P1, it will complete within L_{max}^w time units. At that time, $\mathcal{R}_i^{w,nn}$ will be the earliest-timestamped write request for its requested resource. Summing all the bounds just stated, this occurs within $4L_{max}^w + 2L_{max}^r$ time units in the worst case. □

Theorem 3.4, given next, provides the desired acquisition-delay bounds. Together with Theorem 3.3, this theorem implies that all pi-blocking bounds under the RW-RNLP* are $O(1)$.

Theorem 3.4. *Under the RW-RNLP*, the worst-case acquisition delay of a write request $\mathcal{R}_i^w$ is:*
(i) *L_{max}^r time units, if $\mathcal{R}_i^{w,nn}$ is a non-nested request and no nested requests are active while $\mathcal{R}_i^{w,nn}$ is active;*
(ii) *$L_{max}^w + L_{max}^r$ time units, if $\mathcal{R}_i^{w,nn}$ is a non-nested request and no nested write requests are active while $\mathcal{R}_i^{w,nn}$ is active;*
(iii) *$5L_{max}^w + 3L_{max}^r$ time units, if $\mathcal{R}_i^{w,nn}$ is a non-nested request and nested requests may be active while $\mathcal{R}_i^{w,nn}$ is active;*
(iv) *$3L_{max}^w + 2L_{max}^r$ time units, if $\mathcal{R}_i^{w,n}$ is a nested request.*

Proof. In Case (i), by Lemma 3.8 (i), $\mathcal{R}_i^{w,nn}$ will be the earliest-timestamped active write request for its lone requested resource as soon as it is issued. By Lemma 3.6, it will be satisfied within L_{max}^r time units.

In Case (ii), by Lemma 3.8 (i), $\mathcal{R}_i^{w,nn}$ will be the earliest-timestamped active write request for its lone requested resource as soon as it is issued. By Lemma 3.5, it will be satisfied within $L_{max}^w + L_{max}^r$ time units.

In Case (iii), by Lemma 3.8 (ii), $\mathcal{R}_i^{w,nn}$ will be the earliest-timestamped active write request for its lone requested resource within $4L_{max}^w + 2L_{max}^r$ time units. By Lemma 3.5, it will then be satisfied within $L_{max}^w + L_{max}^r$ time units, resulting in a worst-case acquisition delay of $5L_{max}^w + 3L_{max}^r$ time units.

In Case (iv), by Lemma 3.7, $\mathcal{R}_i^{w,n}$ will be the earliest-timestamped active write request for all of its requested resources within $2L_{max}^w + L_{max}^r$ time units. By Lemma 3.5, it is then satisfied within $L_{max}^w + L_{max}^r$ time units, resulting in a worst-case acquisition delay of $3L_{max}^w + 2L_{max}^r$ time units. □

In Section 3.4.1 it is shown that all of the blocking bounds in Theorem 3.4 are *tight*, *i.e.*, scenarios exist in which these exact bounds occur. Note that, by Theorem 3.3 and Theorem 3.4 (i), if non-nested requests are not affected by nested requests, then read and write requests have worst-case pi-blocking bounds of only $L_{max}^w + L_{max}^r$ and L_{max}^r time units, respectively.

3.2.5 The Fast RW-RNLP with the RW-RNLP*

Referring back to the fast RW-RNLP structure in Figure 3.4, notice that all read requests (both nested and non-nested) directly invoke the RW-RNLP*. Furthermore, Rule P3 ensures that at most one non-nested write request and one nested write request per resource accesses the RW-RNLP* at a time.

Because read requests directly invoke the RW-RNLP*, the pi-blocking incurred by them is $O(1)$ in the worst case (L_{max} is considered to be constant), as shown in the following theorem. Thus, Goals (ii) and (iii) above are met for read requests: non-nested requests have contention-sensitive blocking and nested requests have blocking bounds asymptotically the same as under the RW-RNLP. The following theorem also shows that Goals (ii) and (iii) are met for write requests: the pi-blocking incurred by a non-nested write request $\mathcal{R}_i^{w,nn}$ is $O(N_i)$ in the worst case (recall from Section 2.4 that N_i is the contention experienced by request $\mathcal{R}_i$), and the pi-blocking incurred by a nested write request is $O(m)$ in the worst case. As with the R^{3}LP, the fast RW-RNLP with the RW-RNLP* does result in a higher coefficient for blocking of nested write requests, but the fast RW-RNLP outperforms the RW-RNLP in practice, as shown in Section 3.3.

Theorem 3.5. *Under the fast RW-RNLP with the RW-RNLP*, the worst-case acquisition delay for a request $\mathcal{R}_i$ is:*

(i) $L^w_{max} + L^r_{max}$ *time units, if* $\mathcal{R}^r_i$ *is a read request;*

(ii) $N_i \cdot (L^w_{max} + L^r_{max}) + L^r_{max}$ *time units, if* $\mathcal{R}^{w,nn}_i$ *is a non-nested request and no nested requests are active while* $\mathcal{R}^{w,nn}_i$ *is active;*

(iii) $N_i \cdot (2L^w_{max} + L^r_{max}) + L^w_{max} + L^r_{max}$ *time units, if* $\mathcal{R}^{w,nn}_i$ *is a non-nested request and no nested write requests are active while* $\mathcal{R}^{w,nn}_i$ *is active;*

(iv) $N_i \cdot (6L^w_{max} + 3L^r_{max}) + 5L^w_{max} + 3L^r_{max}$ *time units, if* $\mathcal{R}^{w,nn}_i$ *is a non-nested request and nested requests may be active while* $\mathcal{R}^{w,nn}_i$ *is active;*

(v) $(m-1) \cdot (4L^w_{max} + 2L^r_{max}) + 3L^w_{max} + 2L^r_{max}$ *time units, if* $\mathcal{R}^{w,n}_i$ *is a nested request.*

Proof. Each case follows directly from Theorems 3.1, 3.3, and 3.4. □

In a system with only single-writer resources, the RW-RNLP* alone is sufficient; other protocols are not required to arbitrate access between write requests as no write requests will conflict. Thus, Corollary 3.3 can be applied to show that all requests incur $O(1)$ pi-blocking with very low constant factors.

To this point, the RW-RNLP* is fully specified abstractly. What remains is to devise an actual implementation of it with reasonable overhead.

3.2.6 RW-RNLP* Implementation

Of the building blocks used to construct the fast RW-RNLP, the TL and the RNLP have existing implementations (Brandenburg and Anderson, 2010; Ward and Anderson, 2012). Section 3.1 provided an implementation of the R^{3}LP. Thus, it remains to provide an implementation of the RW-RNLP*. Recall that the focus here is on a user-level, spin-based version.

The main challenge in implementing the RW-RNLP* lies in supporting the atomicity assumptions inherent in the rule-based specification. Such assumptions could be supported by encapsulating certain code regions within lock and unlock calls to an underlying mutex. Indeed, this approach was taken in implementing the rules of the RW-RNLP (Ward and Anderson, 2014). While such an approach introduces additional pi-blocking, the protected critical sections are usually very short, so such blocking is considered to be part of the lock and unlock overhead of the protocol being implemented. Avoiding relying on the use of mutex protocols in this way if possible is preferable, especially *categorically precluding* their use in

Listing 3 RW-RNLP* Definitions

```
type res_state: record
    rin, rout: unsigned integer initially 0
    win, wout: unsigned integer initially 0
constant
    RINC 0x100  // reader increment value
    WBITS 0xff  // writer bits in rin
    PRES  0x80  // writer present bit
    PHID  0x7f  // writer phase ID bits
```

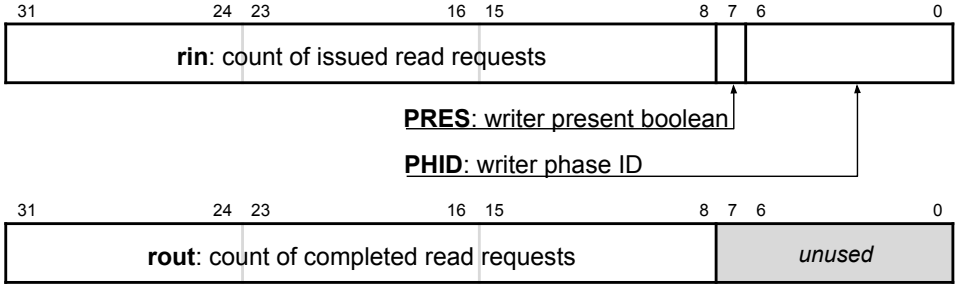


Figure 3.7: Bits in the per-resource *rin* and *rout* variables. (A very similar figure appears in the presentation of PF-TLs (Brandenburg and Anderson, 2010).)

implementing the lock and unlock routines for non-nested requests, as efficiently implementing such routines is the emphasis of this chapter.

The implementation of the RW-RNLP* is based on the same ideas underlying PF-TLs. The shared variables used to track requests are described below and then the pseudocode for each type of request is presented.

Shared variables of the RW-RNLP*. In the implementation, corresponding to each shared resource ℓ_a is a pointer to a structure called *res_state*, which consists of four shared counters, *rin*, *rout*, *win*, and *wout*, as shown in Listing 3. Almost identical counters to these are used in the PF-TL (Brandenburg and Anderson, 2010). Counters *win* and *wout* track the number of write requests for resource ℓ_a that have been issued and completed, respectively. Counters *rin* and *rout* similarly count read requests, with the added complexity of storing information about writes in the bottom byte, as shown in Figure 3.7. Listing 3 shows various constant bitmasks used in the code to access and manipulate certain bits in *rin* and *rout*. As with the R^{3}LP, shared state must be marked `volatile` and updated atomically.

Non-nested requests in the RW-RNLP*. The lock and unlock routines for non-nested requests in the implementation are shown in Listing 4. These are *nearly identical to those for the PF-TL*, which is an efficient reader/writer lock for single-resource requests (Brandenburg and Anderson, 2010). A non-nested read request

Listing 4 RW-RNLP* Routines for Non-Nested Requests

```
 1: procedure R*_LOCK^nn(ℓ: ptr to res_state)
 2:     var w: unsigned int
 3:     w := fetch&add(ℓ→rin, RINC) & WBITS                    ▷ In read queue
 4:     await (w = 0) or (w ≠ (ℓ→rin & WBITS))                 ▷ Satisfied
 5: end procedure

 6: procedure R*_UNLOCK^nn(ℓ: ptr to res_state)
 7:     atomic_add(ℓ→rout, RINC)
 8: end procedure

 9: procedure W*_LOCK^nn(ℓ: ptr to res_state)
10:     var rticket, wticket, w: unsigned int
11:     wticket := fetch&add(ℓ→win, 1)                         ▷ In write queue
12:     await (wticket = ℓ→wout)                               ▷ Head of write queue
13:     w := PRES | (wticket & PHID)
14:     rticket := fetch&add(ℓ→rin, w)                         ▷ Marked entitled now for all reads to see
15:     await (rticket = ℓ→rout)                               ▷ Satisfied
16: end procedure

17: procedure W*_UNLOCK^nn(ℓ: ptr to res_state)
18:     fetch&and(ℓ→rin, ∼(WBITS))                             ▷ Clear WBITS
19:     ℓ→wout := ℓ→wout + 1
20: end procedure
```

$\mathcal{R}_i^{r,nn}$ for a resource ℓ_a is performed by simply incrementing the number of readers for ℓ_a (Line 3) and then spinning if necessary (Line 4). In particular, if ℓ_a is currently being written, then $\mathcal{R}_i^{r,nn}$ waits for a single write request to complete as indicated by either the PRES bit being cleared or the PHID bits being changed, which indicates that a new writer has set those bits, and thus the prior write has completed. To unlock ℓ_a, $\mathcal{R}_i^{r,nn}$ simply increments *rout* by RINC (Line 6).

A non-nested write $\mathcal{R}_i^{w,nn}$ of a resource ℓ_a waits until it holds the earliest ticket among all write requests for ℓ_a (Lines 11–12). It then atomically sets the last byte of ℓ_a's *rin* variable and determines the number of read requests for ℓ_a upon which it must block (Lines 11–14). Next, it waits until those reads (if any) are complete (Line 13). When $\mathcal{R}_i^{w,nn}$ completes, it clears the writer byte of ℓ_a's *rin* variable (Line 15) and increments its *wout* counter (Line 16).

Nested requests in the RW-RNLP*. The lock and unlock routines for nested requests are shown in Listing 5. These routines are very similar to those in Listing 4, with two notable exceptions.

First, an extra phase has been added to the lock routine for read requests (Lines 3–6).[7] The analysis in Section 3.2.4 was based on the assumption that enqueueing takes no time; that is not the case in practice. This extra phase handles a corner case in which unnecessary writer blocking occurs as a result of enqueueing

[7] This extra phase erroneously combined Lines 3–6 into a single loop in the conference paper (Nemitz et al., 2017), given there as Listing 3, but the source code was correct. It has been corrected here.

Listing 5 RW-RNLP* Routines for Nested Requests

```
 1: procedure R*_LOCK^n(D: set of ptr to res_state)
 2:     var w_ℓ: unsigned int for each ℓ in D
 3:     for each ℓ in D:
 4:         w_ℓ := ℓ→rin & WBITS
 5:     for each ℓ in D:
 6:         await (w_ℓ = 0) or (w_ℓ ≠ (ℓ→rin & WBITS))
 7:     R^2LP_LOCK(r_type)
 8:     for each ℓ in D:
 9:         w_ℓ := fetch&add(ℓ→rin, RINC) & WBITS          ▷ Marked entitled for all writes to see
10:     R^2LP_UNLOCK(r_type)
11:     for each ℓ in D:
12:         await (w_ℓ = 0) or (w_ℓ ≠ (ℓ→rin & WBITS))    ▷ Satisfied
13: end procedure

14: procedure R*_UNLOCK^n(D: set of ptr to res_state)
15:     for each ℓ in D:
16:         atomic_add(ℓ→rout, RINC)
17: end procedure

18: procedure W*_LOCK^n(D: set of ptr to res_state)
19:     var rticket_ℓ, wticket_ℓ, w_ℓ: unsigned int for each ℓ in D
20:     for each ℓ in D:
21:         wticket_ℓ := fetch&add(ℓ→win, 1)               ▷ In write queue
22:         await (wticket_ℓ = ℓ→wout)                     ▷ Head of all requested write queues now
23:     R^2LP_LOCK(w_type)
24:     for each ℓ in D:
25:         w_ℓ := PRES | (wticket_ℓ & PHID)
26:         rticket_ℓ := fetch&add(ℓ→rin, w_ℓ)             ▷ Marked entitled for all reads to see
27:     R^2LP_UNLOCK(w_type)
28:     for each ℓ in D:
29:         await (rticket_ℓ = ℓ→rout)                     ▷ Satisfied
30: end procedure

31: procedure W*_UNLOCK^n(D: set of ptr to res_state)
32:     for each ℓ in D:
33:         fetch&and(ℓ→rin, ∼(WBITS))                     ▷ Clear WBITS
34:         ℓ→wout := ℓ→wout + 1
35: end procedure
```

taking some time; this corner case is explored in Section 3.4.2. This extra phase does add an additional $L^w_{max} + L^r_{max}$ time units to the acquisition delay for nested read requests. This additional blocking is accounted for in the schedulability study in Section 3.3.2.

Second, because requests are now for *sets* of resources, such sets must be ensured to atomically enqueue to prevent potential deadlock. (This is why, as discussed in Chapter 2, resources must be acquired according to a predetermined order in the variant of the RNLP that does not use DGLs.) However, it turns out that the only potential deadlock situation that can occur involves a race condition between nested readers and nested writers. Furthermore, this race condition can be eliminated by coordinating access to the lock state with a reader/reader locking protocol (R^{2}LP). In this application of the R^{2}LP, phases are defined such that read

requests are allowed to execute together and write requests may execute together, but read requests and write requests are prevented from executing simultaneously. The calls to the R^{2}LP lock and unlock routines in Lines 7, 10, 23, and 27 specify their type as an input parameter. While using the R^{2}LP introduces blocking overhead, this overhead is only $O(1)$ (Nemitz et al., 2018). This is preferable to the blocking overhead that would result from using a mutex lock to prevent race conditions.

Clearly, the routines in the above implementation are not actually atomic: each executes over a duration of time, not instantaneously. However, it can be formally shown that each routine is linearizable. That is, for each routine, an instantaneous *linearization point* can be defined at which the routine "appears" to take effect atomically (see Section 3.4.3). When viewed in this way, the routines can be shown to support the rule-based specification of the RW-RNLP* given earlier.

3.3 Evaluation

In this section, both variants of the fast RW-RNLP are evaluated on the basis of overhead and blocking via user-space experiments. Additionally, a large-scale overhead-aware schedulability study explores the impact of the worst-case acquisition delays from Theorems 3.2 and 3.5 on system schedulability.

3.3.1 Overhead and Blocking

Given the focus of this chapter, overhead and blocking times for non-nested and read requests are especially relevant. User-space evaluation was conducted on a dual-socket, 18-cores-per-socket, 2.3 GHz Intel Xeon E5-2699 platform running Ubuntu 14.04.

A number of experimental parameters were varied in the evaluation, including the number of tasks (n), nesting depth ($\mathbb{D} = |D_i|$), critical-section length (L_i), probability of a request being nested (rather than non-nested), and probability of a request being a read request (rather than a write request). The following parameter ranges were considered: $n \in \{2,4,...,36\}$, $\mathbb{D} \in \{1,2,...,10\}$, $L_i \in \{0\mu s, 10\mu s, ..., , 100\mu s\}$, and nested and read probabilities independently in $\{0.0, 0.1, ..., 1.0\}$. A scenario is defined as choosing a value for four of the parameters, and varying the fifth. Each task was pinned to a single core, and for task counts of up to 18, all tasks were assigned to the same socket. To simulate behavior that would generate the worst-case lock overhead and blocking times, each task was configured to issue lock and unlock calls 10,000 times, as fast as possible. Each such lock-unlock call pair corresponded to a single request that was randomly chosen

to be nested (or non-nested) and a read (or a write) given the scenario's parameters, for 1 or $\mathbb{D}$ resources randomly chosen from $n_r = 64$ possible resources for non-nested and nested requests, respectively.

In all of the graphs, worst-case values are shown, which were obtained by computing the 99^{th} percentile of all recorded results in order to filter out any spurious measurements (the measurements were taken at user level, so there is no other means for filtering results impacted by interrupts). In the course of the experiments, hundreds of graphs were produced. The protocol implementations and the full set of graphs can be found online.[8]

Overhead and blocking. The considered protocols are compared on the basis of overhead and blocking: recall from Section 1.1 that the *overhead* incurred by a resource request is the total time spent by it executing lock logic within lock and unlock routines (including any time spent waiting to access underlying locks used to enforce atomicity properties required by that logic); the *blocking* incurred by the request is the total time spent by it waiting to access its requested resources. Both overhead and blocking were measured for a number of different scenarios, using the experimental parameters defined above.

The design goals for both fast RW-RNLP variants were to ensure that **(i)** read and non-nested write requests have low overhead and experience contention-sensitive pi-blocking and **(ii)** nested write requests experience pi-blocking that is no worse (and hopefully better) than that under the RW-RNLP. Accordingly, the use of per-resource PF-TLs (which exhibit very low overhead and are contention-sensitive) were used as standards for comparison in assessing (i) and the RW-RNLP (of course) in assessing (ii). A few graphs that are exemplars of trends seen generally are discussed in the following observations.

Observation 3.1. For non-nested read requests (respectively, non-nested write requests), the fast RW-RNLP with the RW-RNLP* and PF-TLs exhibited comparable overhead (respectively, higher overhead).

This observation is supported by Figure 3.8 (a), which plots overhead for both reads and writes under both the fast RW-RNLP and PF-TLs as a function of the task count, n. The data in this figure corresponds to a scenario in which all requests were non-nested, evenly distributed between read and write requests, and the total number of resources, n_r, was set to 64. The critical section of each request was configured to have a duration of 40μs. For comparison, overhead for both protocols held steady in the range of around 0.1μs to 0.5μs for up to 18 tasks, with the fast RW-RNLP with the RW-RNLP* having a higher write-lock overhead than PF-TLs. Implementation-wise, the difference for write requests under the fast RW-RNLP with the

[8] See `https://www.cs.unc.edu/~jarretc/dissertation/`.

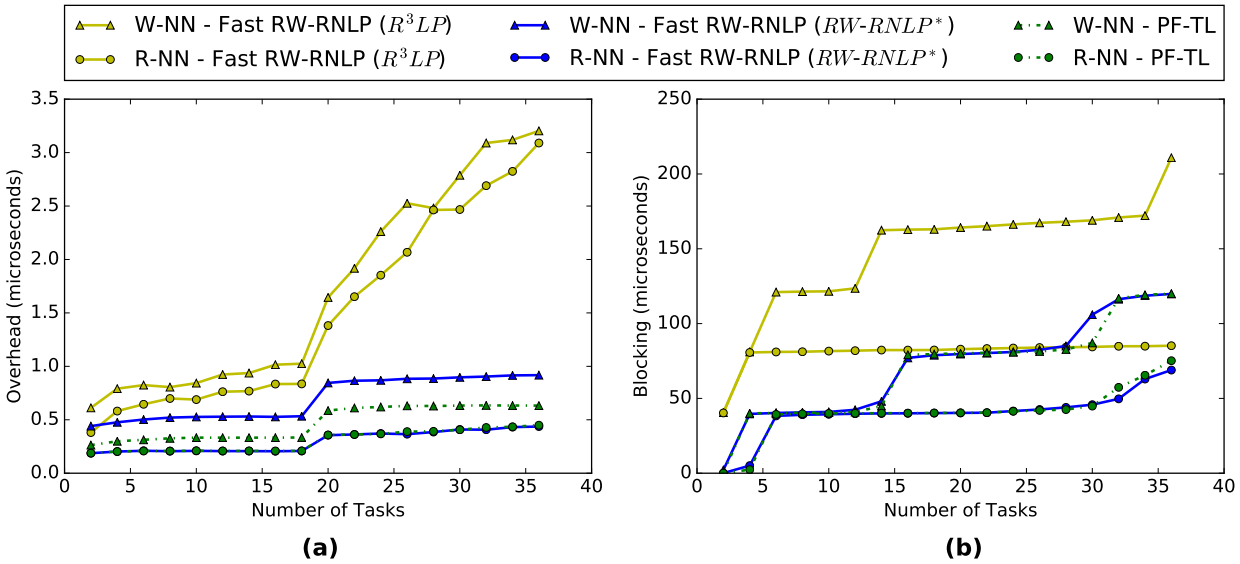


Figure 3.8: **(a)** Overhead and **(b)** blocking for non-nested read and write requests when using PF-TLs versus both variants of the fast RW-RNLP. For each request $\mathcal{R}_i$, $L_i^r = 40\mu$s, $L_i^w = 40\mu$s, $n_r = 64$, $|D_i| = 1$. Requests were randomly chosen to be a read (or a write) with probability 0.5.

RW-RNLP* is that each request must first acquire the ticket lock corresponding to its required resource; this contributes additional overhead (if not also additional blocking). Beyond 18 tasks, overhead increased under both protocols. This is because, beyond a task count of 18, tasks execute on both sockets of the considered platform.

Observation 3.2. The fast RW-RNLP with the R^{3}LP exhibited higher overhead than PF-TLs and the fast RW-RNLP with the RW-RNLP*.

This trend is seen in Figure 3.8 (a) and is unsurprising; unlike the other protocols, the fast RW-RNLP with the R^{3}LP requires all requests to modify parts of the lock state based on request type rather than on a per-resource basis. Therefore, in a system with more resources than request types, the R^{3}LP approach is likely to cause more cache invalidations, in turn causing higher overhead.

Observation 3.3. In general, overhead increased when using two sockets instead of one.

This trend is seen in Figure 3.8 (a), discussed earlier, and also in Figure 3.9 (a) and (b), considered in detail below. When tasks execute on two sockets instead of one, overhead due to maintaining cache coherency increases. Observe that, in Figure 3.8 (a), overhead under the fast RW-RNLP with the RW-RNLP* is never more than around 1.0μs. This value is quite small compared to the 40μs critical-section length.

Table 3.1: Implementation-based worst-case acquisition delay under the fast RW-RNLP.

Request	Case	with the R^{3}LP	with the RW-RNLP*
$\mathcal{R}^r$	2	$L^w + L^r$	$L^w + L^r$
$\mathcal{R}^r$	1	$2L^w + L^r$	$2L^w + 2L^r$
$\mathcal{R}^{w,nn}$	2	$N_i \cdot (2L^w + L^r) + L^w + L^r$	$N_i \cdot (L^w + L^r) + L^r$
$\mathcal{R}^{w,nn}$	3	$N_i \cdot (2L^w + L^r) + L^w + L^r$	$N_i \cdot (2L^w + L^r) + L^w + L^r$
$\mathcal{R}^{w,nn}$	1	$N_i \cdot (3L^w + L^r) + 2L^w + L^r$	$N_i \cdot (6L^w + 3L^r) + 5L^w + 3L^r$
$\mathcal{R}^{w,n}$	1	$(m-1) \cdot (3L^w + L^r) + 2L^w + L^r$	$(m-1) \cdot (4L^w + 2L^r) + 3L^w + 2L^r$

Cases: [1] No restrictions [2] No nested requests [3] No nested write requests
Note that, for brevity, L^w (respectively, L^r) is used here to denote L^w_{max} (respectively, L^r_{max}).
For reference, the bounds of RW-RNLP: $L^w + L^r$ for $\mathcal{R}^r$ and $(m-1)(L^w + L^r)$ for $\mathcal{R}^w$.

For the RW-RNLP with the R^{3}LP, the overhead is as high as 3.2μs, but still significantly lower than the critical-section length.

Observation 3.4. In scenarios with only non-nested requests, the fast RW-RNLP with the RW-RNLP* and PF-TLs exhibited nearly identical blocking.

This observation is clearly supported by Figure 3.8 (b). Together with Observation 1, this observation suggests the viability of providing the fast RW-RNLP with the RW-RNLP* as a general synchronization solution. It can be used in systems in which nested requests do not occur with no detrimental impacts of note.

Observation 3.5. In general, the fast RW-RNLP with the R^{3}LP exhibited higher observed blocking than either PF-TLs or the fast RW-RNLP with the RW-RNLP*.

With requests of each type present, the R^{3}LP cycles between phases in a manner that can easily cause the worst-case acquisition delay to be experienced by requests. This is in contrast to the fast RW-RNLP with the RW-RNLP*, which requires conflicting requests to be issued in precisely the worst order to actually realize the worst-case blocking. This may be because the particular request issuance order required to generate the worst-case did not occur during the experiments. Recall, however, that the R^{3}LP has lower analytical worst-case blocking bounds, as shown in Sections 3.2.2 and 3.2.5 and summarized in Table 3.1.

Observation 3.6. In scenarios with both nested and non-nested requests, overhead for write requests tended to be much lower under both fast RW-RNLP variants than under the RW-RNLP.

This observation is supported by Figure 3.9 (a) and (b), which depict data from two different scenarios, as detailed in the figure's caption. The higher overhead under the RW-RNLP is partially due to the use of write expansion (recall Figure 3.6), which increases resource contention. This increased contention impacts

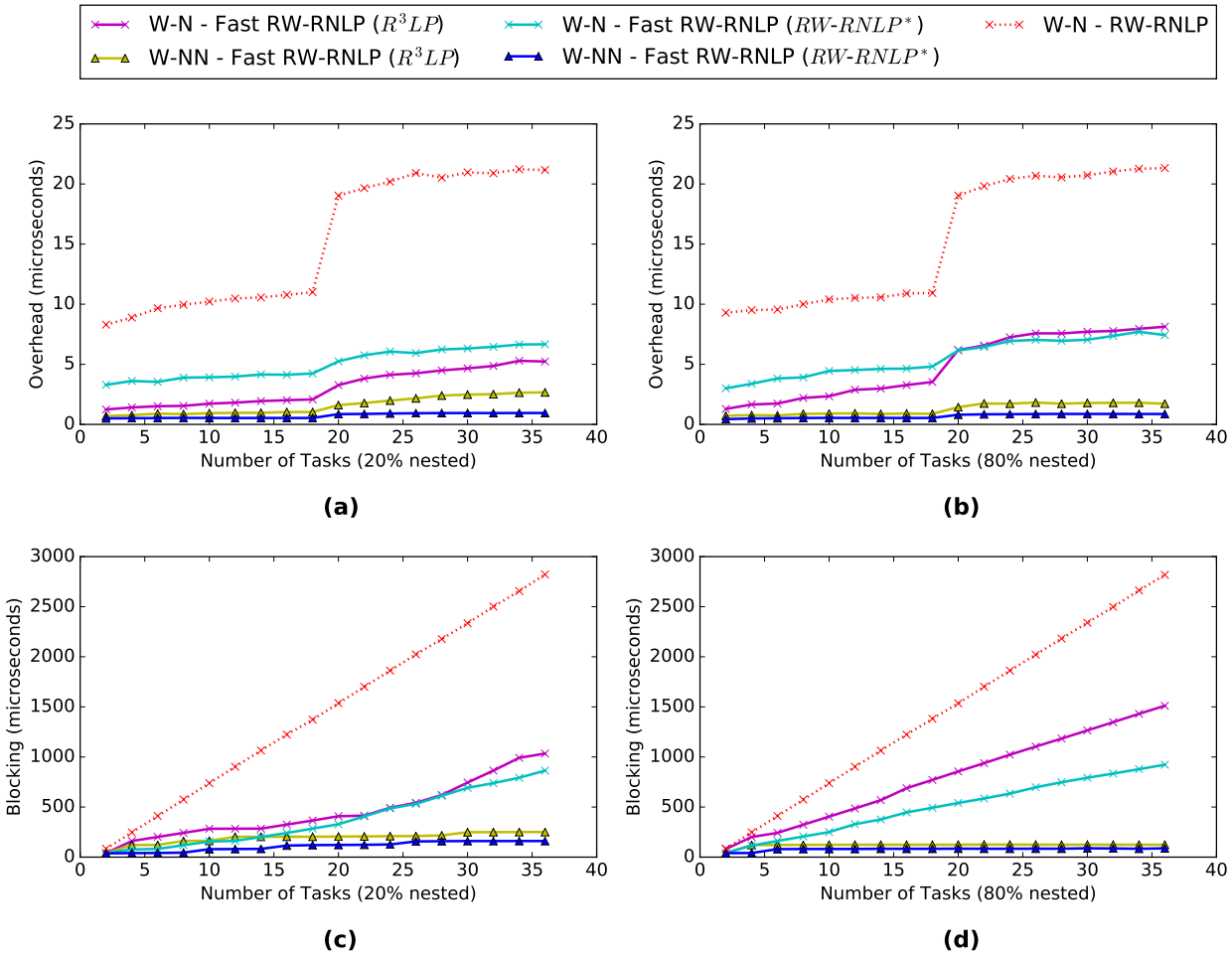


Figure 3.9: **(a), (b)** Overhead and **(c), (d)** blocking for nested and non-nested write requests under the RW-RNLP and the fast RW-RNLP. Here, $L_i^r = 40\mu$s, $L_i^w = 40\mu$s, $n_r = 64$, $|D_i| = 1$, for non-nested requests, and $|D_i| = 4$, for nested requests. Requests were chosen to be a read (or write) with probability 0.5. Data is plotted for the cases of 20% (left) and 80% (right) of requests being nested. Due to write expansion (recall Figure 3.6), D_i was inflated to include all 64 resources for writes under the RW-RNLP.

the overhead of write requests, as they write-lock an underlying PF-TL to update all relevant resource queues atomically (Ward and Anderson, 2014). Note that, under the RW-RNLP, write expansion forces non-nested write requests to be processed like nested ones.

Observation 3.7. In scenarios with both nested and non-nested requests, blocking for write requests tended to be much lower under both fast RW-RNLP variants than under the RW-RNLP.

This observation is supported by Figure 3.9 (c) and (d), which plot recorded worst-case blocking times associated with the scenarios in Figure 3.9 (a) and (b). For $m = 36$, blocking was up to 18 times lower (respectively, 12 times lower) under the fast RW-RNLP with the RW-RNLP* (respectively, with the R^{3}LP)

than under the RW-RNLP; write expansion increases resource contention, which increases blocking times of the RW-RNLP.

Observation 3.8. Non-nested write requests exhibited contention-sensitive blocking under the fast RW-RNLP variants but not the RW-RNLP.

This observation is also supported by Figure 3.9 (c) and (d). Notice that, as the task count increases, the potential for additional blocking increases due to transitive blocking, which negatively impacts any protocol that provides no mechanisms for eliminating transitive blocking. Blocking for non-nested requests under the fast RW-RNLP increases slowly as the task count increases; with more tasks, more contention is possible, and a slow linear growth of contention (and thus blocking) with the number of tasks is expected. In contrast, non-nested write requests are converted to nested ones under the RW-RNLP due to write expansion. As a result, their blocking under that protocol is not $O(N)$, but instead $O(m)$. This translates to a faster linear growth of blocking, as in Figure 3.9 (c) and (d).

Notice that Figure 3.9 pertains to write requests. The corresponding read request results are shown in Figure 3.10. Both overhead and blocking were much lower for reads than for writes, as expected. Under the fast RW-RNLP variants, non-nested read requests had higher blocking than under the RW-RNLP by 1-2 critical-section lengths, and nested read requests had higher blocking by 2-3 critical-section lengths, as expected from the implementation-based worst-case acquisition delay bounds in Table 3.1.

Of relevance to the analysis presented in Section 3.2, Figure 3.11 demonstrates the results of varying the critical-section length while holding the number of tasks n constant (in these experiments, m and n are equal). In contrast, in Figure 3.9 (b) the number of tasks was varied, and the critical-section length was held constant; the points in Figure 3.9 (c) at $m = 36$ are the same as those in Figure 3.11 for $L_i = 40\mu$s. Note that varying m effectively modifies the term N_i for each request $\mathcal{R}_i$.

Observation 3.9. Blocking time scaled linearly with critical-section length for both the fast RW-RNLP variants and the RW-RNLP.

Figure 3.11 illustrates this observation, which reflects expected behavior based on the blocking analysis; for each type of request, the worst-case blocking bound contains both L^w_{max} and L^r_{max} terms with different coefficients depending on the request type.

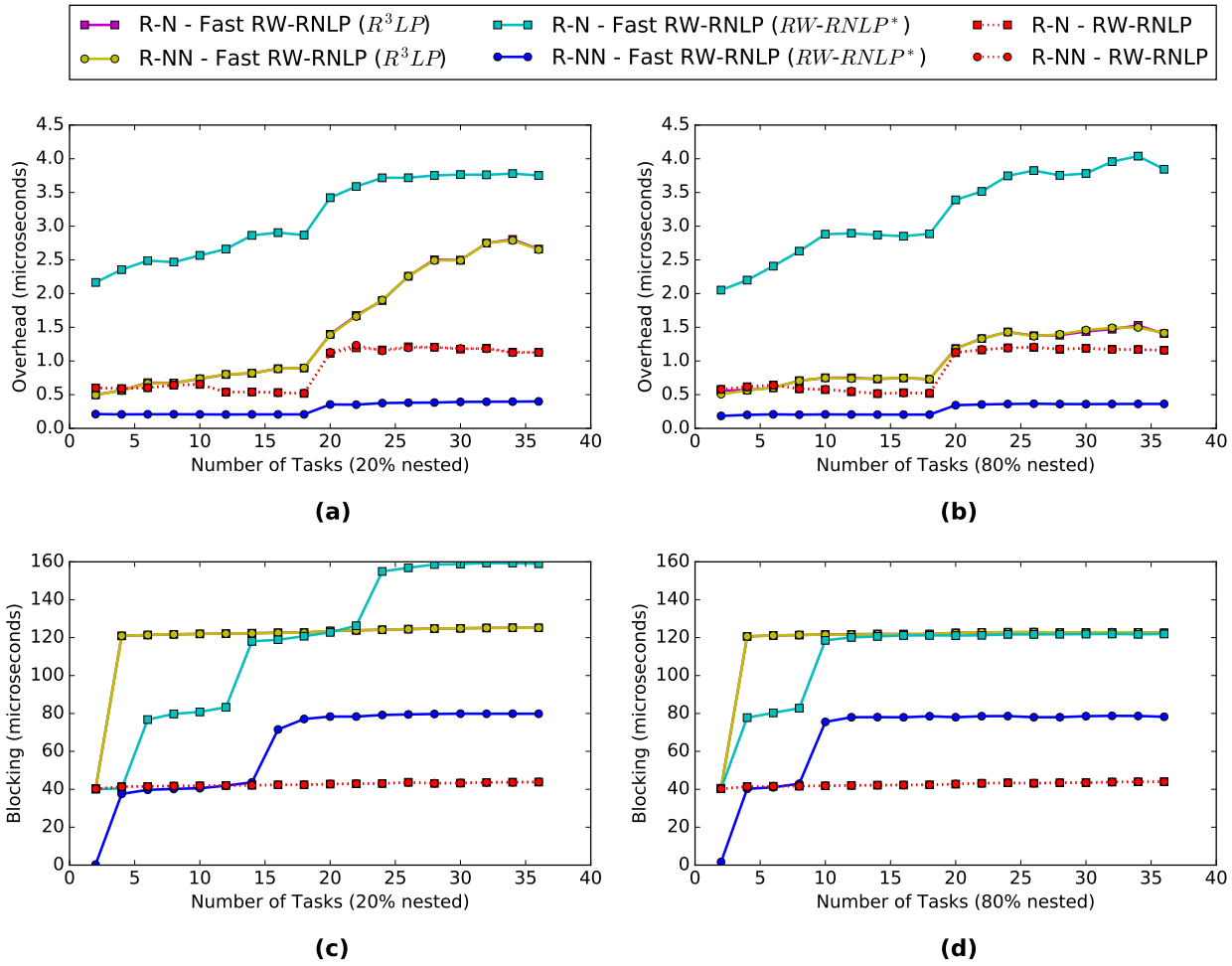


Figure 3.10: **(a), (b)** Overhead and **(c), (d)** blocking for nested and non-nested read requests under the RW-RNLP and the fast RW-RNLP, in the same scenario as in Figure 3.9.

Though the fast RW-RNLP results in higher coefficients for nested write requests than the bounds proven for the RW-RNLP, lower blocking times were generally seen under both variants of the fast RW-RNLP. This difference may be because, under the RW-RNLP, write expansion guarantees that all write requests conflict.

There were also differences between nested and non-nested write requests under the fast RW-RNLP variants, highlighting the improvement of $O(N)$ over $O(m)$ blocking. Under the fast RW-RNLP, the $O(N)$ blocking of non-nested write requests was almost identical to the $O(1)$ blocking of nested read requests. Thus, there is a significant benefit that can be gained when contention is guaranteed to be low.

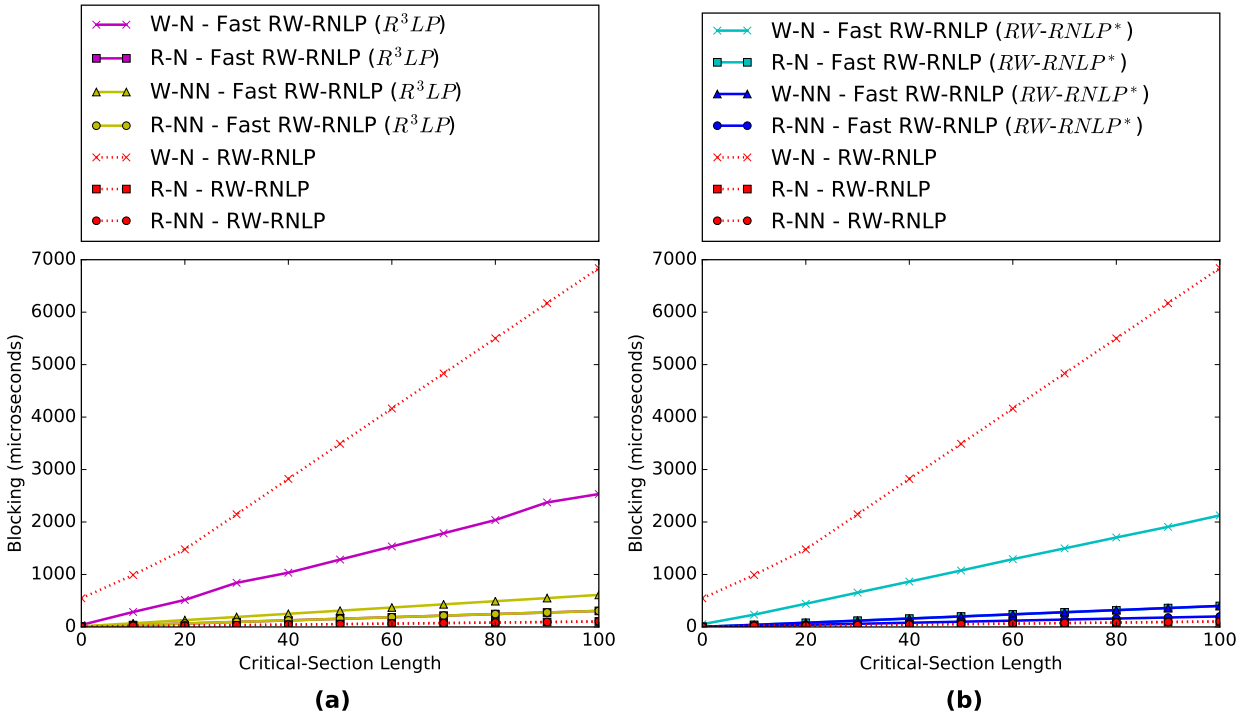


Figure 3.11: Blocking for nested and non-nested write requests under the RW-RNLP and the fast RW-RNLP. The critical-section length varies, $m = 36$, $n_r = 64$, $|D_i| = 1$, for non-nested requests, and $|D_i| = 4$, for nested requests. ($|D_i|$ is inflated to 64 under the RW-RNLP as above.) A request was chosen to be a write with probability 0.5.

3.3.2 Schedulability study

The results presented in Section 3.3.1 demonstrate the tradeoffs between protocols in experimentally measured overhead and blocking times. This section presents an evaluation of the two fast RW-RNLP variants on the basis of hard real-time schedulability. The large-scale study presented here varied a range of parameters, detailed below, and took over 100 CPU-days on the platform described above.

This presentation begins by introducing several additional constraints that can be applied to tighten the computed blocking. Each protocol analyzed is then discussed. Finally, the range of the schedulability study and key findings are discussed.

Constraints. For each task system, blocking was calculated using the worst-case acquisition delay bounds presented in Table 3.1. However, these bounds were tightened using several constraints. Instead of accounting for the system-wide worst-case critical section as repeatedly causing L_{max} blocking for other requests, *period-based constraints* were imposed that limit the number of times each critical section can delay a given request

based on the period of each task. Based on the functionality of each protocol, if two write requests share a resource queue, the FIFO nature of the protocol enforces that each such write can delay the write request of interest at most once. This constraint on blocking is called the *FIFO constraint*. Similarly, the number of critical sections of read requests is limited by the number of write requests that can be counted. This is referred to as the *read-write constraint*. Finally, the blocking of non-nested write requests in the fast RW-RNLP variants depends on contention. Because the only contending requests that impact blocking are other non-nested write requests (recall Theorem 3.1), that number of requests is used for the *contention constraint*. More details and examples of these constraints can be found in Section 3.4.4.

Protocols evaluated. Four protocols were evaluated in this study: the PF-TL, the RW-RNLP, the fast RW-RNLP with the R^3LP, and the fast RW-RNLP with the RW-RNLP*. The latter three protocols are as described above, and the PF-TL is applied to protect the group of all resources; that is, all resources are statically grouped, and this group is protected with a standard PF-TL, eliminating all nesting with the coarse-grained grouping. This application of a PF-TL is referred to as a group PF-TL.

Experimental setup. SchedCAT (SchedCAT, 2019), an open-source real-time schedulability test toolkit, was used to randomly generate task systems, implement blocking bound computations, and check for schedulability on an 18-core platform with global EDF scheduling. A wide range of system parameters were varied, and for each set, task sets were generated with system utilizations in $\{2.0, 2.5, ..., 18.0\}$. Tasks sets with *short* ($[3, 33]$ms), *moderate* ($[10, 100]$ms), and *long* ($[50, 250]$ms) task periods were examined. For each of these ranges, per-task utilizations were varied between *medium* (uniformly chosen from $[0.1, 0.4]$) and *heavy* (uniformly chosen from $[0.5, 0.9]$). Tasks were chosen to issue a single request with a probability chosen from $\{0.1, 0.2, 0.5, 1.0\}$. Each request was a read (as opposed to a write) request with a probability in $\{0.0, 0.2, 0.5, 0.8\}$ and nested (as opposed to non-nested) with a probability in $\{0.01, 0.05, 0.1, 0.2, 0.5\}$. Nested requests were all for four resources chosen randomly from a set of 64 resources. The critical-section length for each request was chosen uniformly within *short* ($[1, 15]\mu$s) or *long* ($[100, 1000]\mu$s).

For each generated task system, the impact of blocking on each request was computed given the constraints discussed above. Most types of overhead were ignored in this evaluation, such as migration overhead and other overhead sources that impact each scheme similarly. However, the overhead incurred by using a specific locking protocol was accounted for; the appropriate overhead values for nested and non-nested read and write requests were applied based on the experimental results presented in Section 3.3.1.

The maximum values of lock and unlock overhead from relevant scenarios was chosen (eliminating scenarios with 100% read requests and rerunning each protocol evaluation for a critical-section length of 1μs, the shortest critical section used in the schedulability study). For the fast RW-RNLP with the RW-RNLP*, the worst-case overhead values for short critical-section lengths were significantly higher than that for medium or long critical-section lengths, so the correct overhead measurement was applied based on the given scenario. For all other protocols, overhead values were chosen only based on request type (*e.g.*, non-nested read).

The schedulability experiments resulted in 960 plots.[9] The following figures and tables highlight a few interesting trends.

Schedulability versus nested probability. In Figure 3.12, each plot shows the schedulability curve of each protocol; a point on a given curve indicates that, given the system utilization shown, the corresponding fraction of task systems generated for this scenario were deemed schedulable. After applying the appropriate blocking bounds and protocol overhead values to each task, schedulability was checked with the tests described in Chapter 2, culminating with Baruah's G-EDF schedulability test (Baruah, 2007). For each point, between 1,000 and 100,000 task systems were generated at random from within the specified ranges. The line denoted NOLOCK shows the fraction of task systems for a given utilization that were schedulable when no additional interference was caused by non-preemptive critical sections or non-preemptive spin blocking.

For the plots shown in Figure 3.12, tasks systems were generated that had medium task utilization, long periods, long critical sections, and with all tasks issuing resource requests. Requests were chosen to be read requests with probability 0.8. Each of the subplots shows the schedulability results given a different percent of nested requests.

In addition to highlighting key trends in the figures, this section presents data summarizing all results. For each of the 960 graphs, the *schedulable utilization area* (SUA) of each protocol was computed; this is the area under the curve for that protocol as approximated by a midpoint Riemann sum. In general, a higher SUA indicates better schedulability. Table 3.2 presents a breakdown of the number of times each protocol was the best (in terms of SUA) by scenario. All scenarios in which all four protocols performed equally (within 2% of each other) were filtered out. The highest entry per nested probability is shown in bold.

Observation 3.10. For task systems in which non-nested requests are the common case, the fast RW-RNLP with the R^{3}LP outperformed the RW-RNLP and the group PF-TL.

[9]These plots are available at `https://www.cs.unc.edu/~jarretc/dissertation/`.

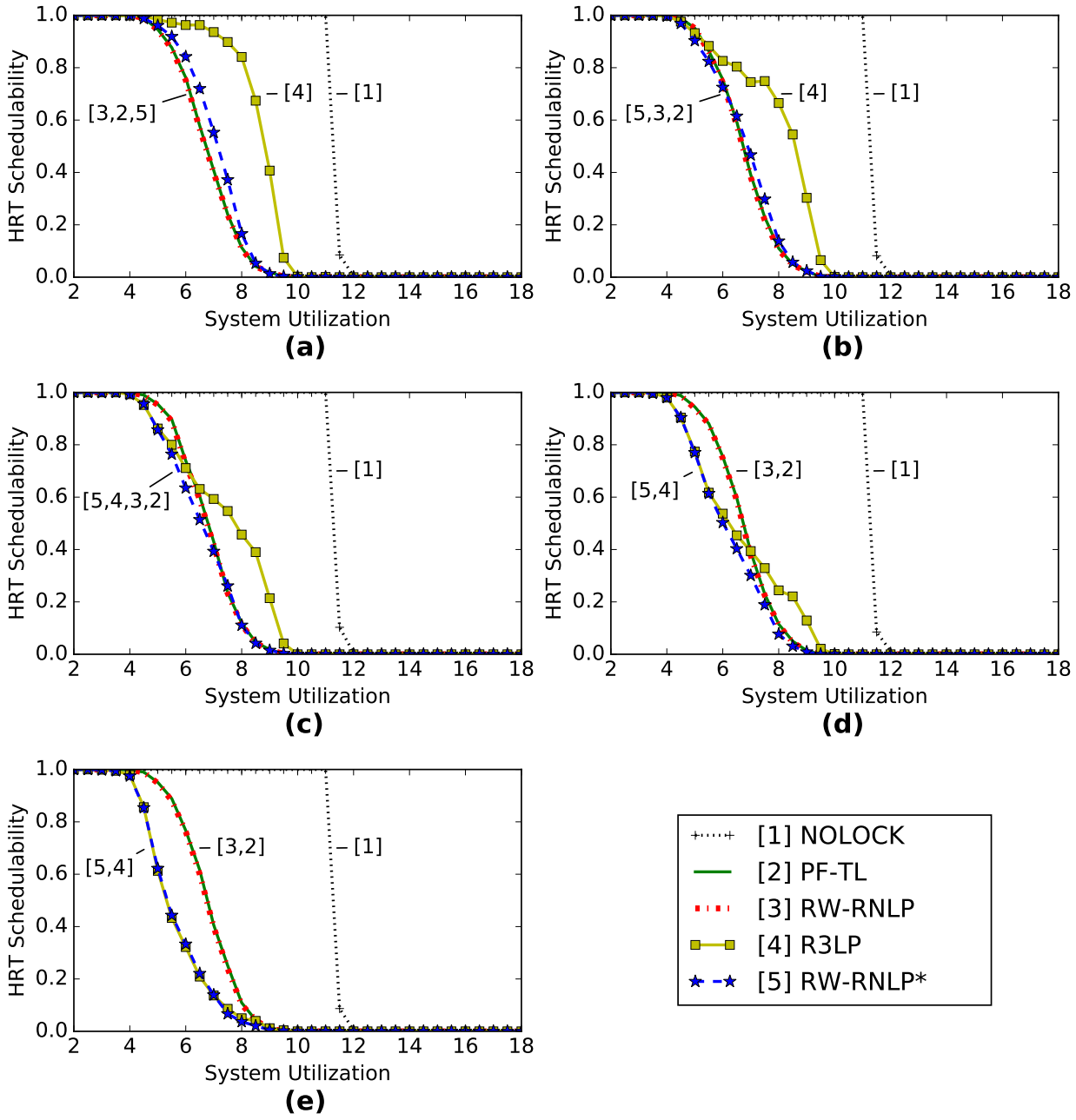


Figure 3.12: Hard real-time schedulability results with varying nested probabilities for the scenario with medium task utilizations, long periods, long critical-section lengths, and read probability 0.8. Nested probabilities are **(a)** 0.01, **(b)** 0.05, **(c)** 0.1, **(d)** 0.2, and **(e)** 0.5.

Figure 3.12 (a), (b), and (c) reflect the observed trends as the percentage of requests which are nested varies. When only 1%, 5%, or 10% of requests are nested, the fast RW-RNLP variant with R^3LP tended to perform as well or better than the two existing protocols. This trend is highlighted in Table 3.2.

Table 3.2: Best protocols per scenario by SUA.

Task Util.	Nested Prob.	PF-TL	RW-RNLP	R^3LP	RW-RNLP*	All tied
medium	0.01	15	1	**75**	13	18
	0.05	28	5	**75**	14	18
	0.1	44	10	**68**	10	18
	0.2	**66**	30	31	5	17
	0.5	**76**	35	0	5	17
heavy	0.01	7	1	**30**	6	63
	0.05	11	5	**28**	4	64
	0.1	13	8	**25**	3	64
	0.2	**19**	11	12	4	64
	0.5	**33**	18	2	5	59

Number of scenarios with medium (top) and heavy (bottom) task utilizations in which each protocol had the highest SUA. Each line contains 96 total scenarios, and any protocols within 2% of the highest SUA for that scenario was also counted as the best.

Observation 3.11. For most task systems explored in which 50% of requests were nested, the group PF-TL and RW-RNLP outperformed the other protocols.

This is reflected in Figure 3.12 (e) and quantified in Table 3.2.

Schedulability versus task utilization and period. In Figure 3.13, schedulability curves for each protocol are shown for task systems with short critical-section lengths, nested probability 0.1, read probability 0.2, and all tasks making requests.

Observation 3.12. The fast RW-RNLP with the R^3LP resulted in higher schedulability than the fast RW-RNLP with the RW-RNLP* in most task systems.

This trend can be observed in Figure 3.12 and Figure 3.13 (a) and (b). In some task systems, the fast RW-RNLP variants displayed almost identical schedulability (as shown in Figure 3.13 (c)), and in very few task systems did the RW-RNLP* variant outperform the R^3LP variant. This trend is also reflected in Table 3.2, in which the R^3LP resulted in the highest SUA more often than the RW-RNLP*.

To compare groups of scenarios, the SUA of all scenarios in the group is summed. Table 3.3 presents the ratio of each of these compared to NOLOCK. This serves to give some intuition about the impact of shared resources managed by each protocol on schedulability with respect to the schedulability when no resource management is required. As before, the highest entry per nested probability is bolded.

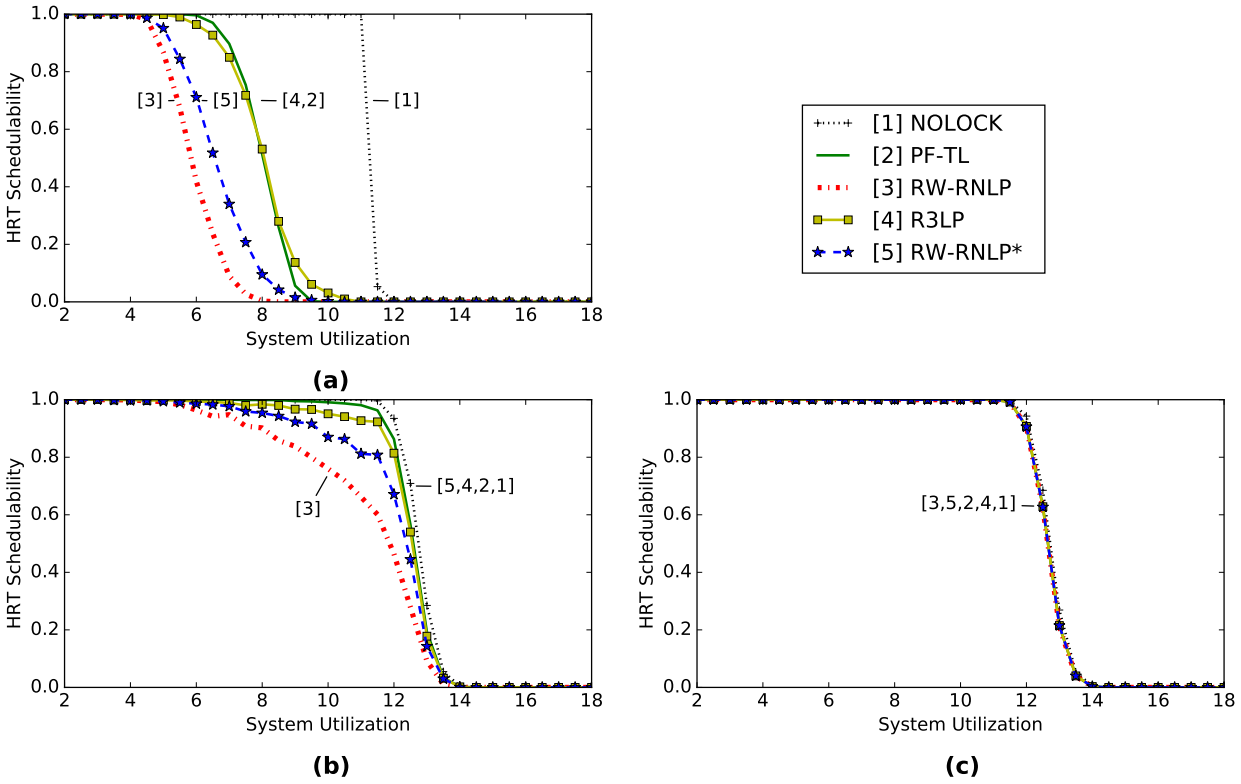


Figure 3.13: Hard real-time schedulability results with varying task utilizations and periods for the scenario with short critical-section lengths, nested probability 0.1, and read probability 0.2. Task utilizations and periods are, respectively, **(a)** medium and short, **(b)** heavy and short, and **(c)** heavy and long.

Observation 3.13. Given a scenario, changing from medium task utilization to heavy task utilization tended to make all protocols have higher schedulability.

This is as expected; with each task having a higher utilization, generally fewer tasks (and thus possible requests) are necessary to hit each utilization threshold. This is supported by Figure 3.13 (a) and (b), as well as by Table 3.3. In Table 3.3, all relative SUAs increase when the task utilization changes from medium to heavy.

Observation 3.14. For some task systems, the locking protocol chosen had a minimal effect on schedulability.

For task systems with medium task utilizations, approximately one-sixth of all scenarios resulted in identical schedulability (within 2% difference) for each locking protocol considered. This effect is also visible in Figure 3.13 (c). In scenarios with heavy task utilization, this effect is even more pronounced, with approximately two-thirds of the scenarios having identical schedulability (Table 3.2).

Table 3.3: Relative SUAs.

Task Util.	Nested Prob.	PF-TL	RW-RNLP	R^{3}LP	RW-RNLP*
medium	0.01	0.685	0.656	**0.748**	0.670
	0.05	0.685	0.656	**0.719**	0.655
	0.1	0.685	0.656	**0.695**	0.643
	0.2	**0.685**	0.656	0.664	0.627
	0.5	**0.685**	0.656	0.620	0.601
heavy	0.01	0.830	0.825	**0.865**	0.815
	0.05	0.830	0.825	**0.852**	0.811
	0.1	0.830	0.825	**0.839**	0.806
	0.2	**0.830**	0.825	0.822	0.800
	0.5	**0.830**	0.825	0.796	0.788

Fraction of summed SUA for each protocol relative to the summed SUA of NOLOCK.

3.4 Additional Details

This section provides additional details on several claims made earlier in the chapter.

3.4.1 Tight Blocking Bounds for the RW-RNLP*

To show that each blocking bound proven for the RW-RNLP* is tight, the section illustrates that each worst-case bound can actually occur by means of examples. An example corresponding to each lemma and theorem about the RW-RNLP* is presented below in the order in which the lemmas and theorems appear in Section 3.2.4. In each example, requests are numbered in the order in which they were issued.

Lemma 3.4 bounds the acquisition delay that a write request can experience after becoming entitled to L^r_{max}.

Example 3.5. As shown in Figure 3.14, write request $\mathcal{R}^w_2$, issued just after $\mathcal{R}^r_1$, is immediately entitled and can experience L^r_{max} acquisition delay. This is exactly the upper bound presented in Lemma 3.4. ◊

Theorem 3.3 bounds the acquisition delay a read request can experience to $L^w_{max} + L^r_{max}$.

Example 3.6. In Figure 3.14, read request $\mathcal{R}^r_3$ experiences an acquisition delay of up to $L^w_{max} + L^r_{max}$ time units. It was issued after the issuance of requests $\mathcal{R}^r_1$ and $\mathcal{R}^w_2$, all for the same resource. $\mathcal{R}^r_3$ cannot be satisfied

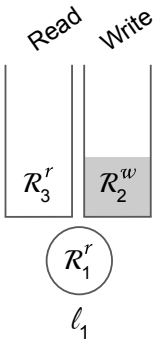


Figure 3.14: A simple example that shows worst-case acquisition delay for a read request and the acquisition delay a write may experience after becoming entitled.

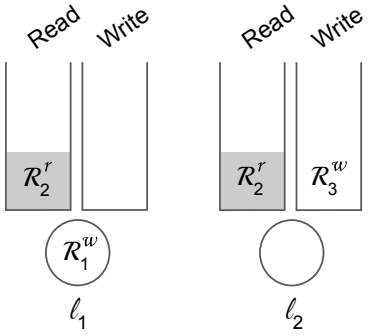


Figure 3.15: An issuance order which may cause the maximum blocking after a write request $\mathcal{R}_3^w$ becomes the earliest-timestamped active write request for each of its resources, here just ℓ_2.

initially, as $\mathcal{R}_2^w$ is entitled. Therefore it waits for up to L_{max}^r time units for $\mathcal{R}_1^r$ to complete. Once $\mathcal{R}_2^w$ is satisfied, $\mathcal{R}_3^r$ waits for up to L_{max}^w time units for $\mathcal{R}_2^w$ to complete before acquiring the resource. $\Diamond$

According to Lemma 3.5, a write request $\mathcal{R}_i^w$ may experience up to $L_{max}^w + L_{max}^r$ blocking after becoming the earliest-timestamped active write request for each resource in D_i.

Example 3.7. Similarly to the previous examples, in Figure 3.15, write request $\mathcal{R}_3^w$ can experience the worst-case delay stated in Lemma 3.5. Because requests were issued in increasing index order, $\mathcal{R}_3^w$ can potentially block for the entire critical sections of $\mathcal{R}_1^w$ and $\mathcal{R}_2^r$, which can be as high as L_{max}^w and L_{max}^r, respectively. $\Diamond$

The earliest-timestamped non-nested write request with no nested requests present can experience blocking of L_{max}^r. This upper bound proven in Lemma 3.6 is shown to be tight in Example 3.5 with $\mathcal{R}_2^w$ as depicted in Figure 3.14.

Lemma 3.7 bounds the time a nested write request must wait before becoming the earliest-timestamped write request for all of its resources to $2L^w_{max} + L^r_{max}$. The following example shows this bound is tight.

Example 3.8. As shown in Figure 3.16 (a), $\mathcal{R}^{w,n}_4$ is not the earliest-timestamped active write request for each of $D_4 = \{\ell_2, \ell_3\}$ when it is issued. In fact, it must wait until $\mathcal{R}^w_3$ has completed. Given that each of these requests could have been issued immediately after each other and that $\mathcal{R}^w_4$ will need to wait until $\mathcal{R}^w_1$, $\mathcal{R}^r_2$, and $\mathcal{R}^w_3$ complete, $\mathcal{R}^w_4$ may wait up to $2L^w_{max} + L^r_{max}$ time units to become the earliest-timestamped active write request. $\Diamond$

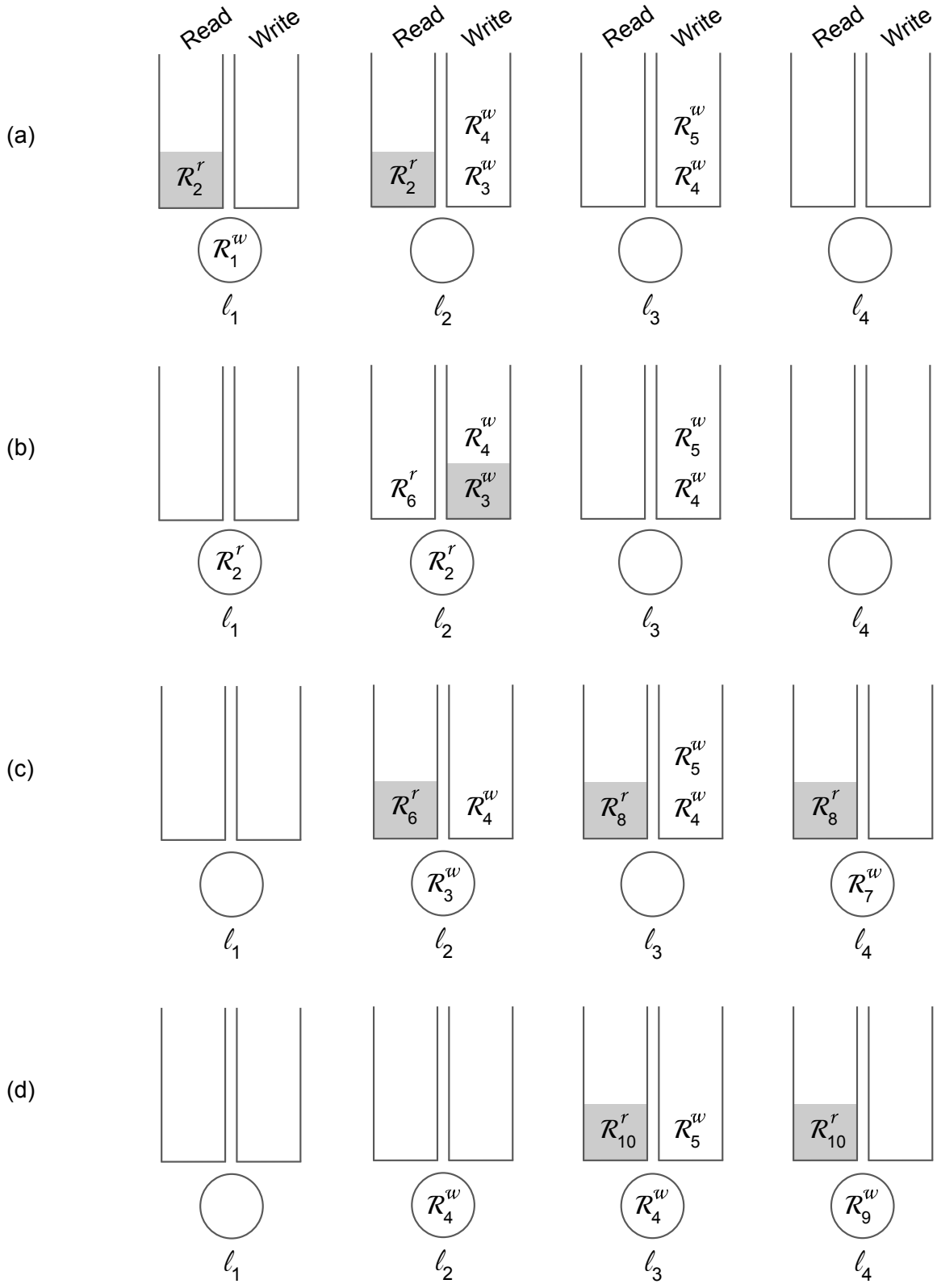


Figure 3.16: A series of read and write requests that illustrate the worst-case acquisition delay for nested and non-nested write requests.

Lemma 3.8 has two cases for how soon a non-nested write request $\mathcal{R}_i^{w,nn}$ will become the earliest-timestamped request for each of its resources. Case (i) does not need an example: the worst-case delay for $\mathcal{R}_i^{w,nn}$ becoming the earliest-timestamped active write request in the RW-RNLP* for D_i is zero when no nested requests are active. Case (ii) bounds this time to $4L_{max}^w + 2L_{max}^r$ in the presence of nested requests.

Example 3.9. Consider $\mathcal{R}_5^w$ in Figure 3.16. This non-nested write request may wait for up to $4L_{max}^w + 2L_{max}^r$ time units to become the earliest-timestamped request for $D_5 = \{\ell_3\}$. To become the earliest-timestamped request for D_5, $\mathcal{R}_5^w$ must wait for $\mathcal{R}_4^w$ to complete, which in turn must wait for $\mathcal{R}_3^w$ to complete. As shown in Figure 3.16 (a), $\mathcal{R}_3^w$ may wait for up to L_{max}^w time units to become entitled (the time for $\mathcal{R}_1^w$ to complete, after which $\mathcal{R}_2^r$ is no longer entitled). After $\mathcal{R}_3^w$ becomes entitled, $\mathcal{R}_6^r$ is issued, as shown in (b). After up to L_{max}^r time units after $\mathcal{R}_3^w$ becomes entitled, $\mathcal{R}_2^r$ completes and $\mathcal{R}_3^w$ is satisfied. $\mathcal{R}_3^w$ may execute for just under L_{max}^w time units before $\mathcal{R}_7^w$ and $\mathcal{R}_8^r$ are issued, as shown in (c). Once $\mathcal{R}_3^w$ completes, $\mathcal{R}_4^w$ may still wait for up to $L_{max}^w + L_{max}^r$ time units before becoming satisfied (for $\mathcal{R}_7^w$ and $\mathcal{R}_8^r$ to complete). After $\mathcal{R}_4^w$ is satisfied, it may execute for L_{max}^w time units, after which $\mathcal{R}_5^w$ is finally the earliest-timestamped request for D_5 after waiting for up to $4L_{max}^w + 2L_{max}^r$ time units. $\Diamond$

Theorem 3.4 presents four bounds for write requests. Non-nested write requests may experience up to L_{max}^r time units of acquisition delay if no nested requests are active (illustrated in Figure 3.14 and described in Example 3.5). If nested read requests may be present but no nested write requests are active, non-nested write requests may experience up to $L_{max}^w + L_{max}^r$ time units of acquisition delay, as illustrated in Figure 3.15 and described in Example 3.7. The third bound presented in Theorem 3.4 is that non-nested write requests in the presence of nested requests may experience up $5L_{max}^w + 3L_{max}^r$ time units of acquisition delay (illustrated below). Finally, nested write requests may experience acquisition delay of up to $3L_{max}^w + 2L_{max}^r$ (also illustrated below).

Example 3.10. As illustrated by Figure 3.16 and Example 3.9, $\mathcal{R}_5^w$ may wait for $4L_{max}^w + 2L_{max}^r$ time units to become the earliest-timestamped request for its resources. Suppose just before $\mathcal{R}_4^w$ completes, $\mathcal{R}_9^w$ and $\mathcal{R}_{10}^r$ are issued, as illustrated in Figure 3.16 (d). (This is similar to the situation in Example 3.9 when $\mathcal{R}_7^w$ and $\mathcal{R}_8^r$ were issued just before the completion of $\mathcal{R}_3^w$.) $\mathcal{R}_5^w$ may indeed need to wait an additional $L_{max}^w + L_{max}^r$ time units before being satisfied, making its total acquisition delay $5L_{max}^w + 3L_{max}^r$ time units.

Figure 3.16 also illustrates that a nested write request, namely $\mathcal{R}_4^w$, may experience acquisition delay of $3L_{max}^w + 2L_{max}^r$. Indeed, $\mathcal{R}_4^w$ waits for the completion of three write requests ($\mathcal{R}_1^w$, $\mathcal{R}_3^w$, and $\mathcal{R}_7^w$), which

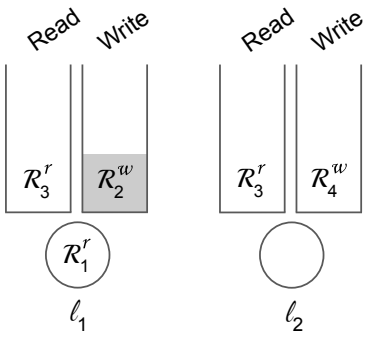


Figure 3.17: Illustrates the edge case in which a write request ($\mathcal{R}_4^w$) would need to wait unnecessarily behind a nested read request ($\mathcal{R}_3^r$) if the extra code step had not been added in Listing 5.

may only barely overlap, and two read phases (those of $\mathcal{R}_2^r$ and $\mathcal{R}_8^r$) that do not overlap with any of the write requests. ◊

3.4.2 Corner Case for Nested Read Requests

If the extra phase in Lines 3-6 of Listing 5 is not included for the R*_LOCK^n routine, a potential edge case exists, as demonstrated in Figure 3.17. In this edge case, write requests suffer unnecessary transitive blocking caused by read requests incorrectly marking themselves entitled.

In this scenario, read request $\mathcal{R}_1^{r,nn}$ is satisfied and write request $\mathcal{R}_2^{w,nn}$ is entitled when read request $\mathcal{R}_3^{r,n}$ is issued. At this point, $\mathcal{R}_1^{r,nn}$ has completed the R*_LOCK^{nn} routine and $\mathcal{R}_2^{w,nn}$ is waiting at Line 15 for its requested resource to become available.

Without Lines 3-6, $\mathcal{R}_3^{r,n}$ immediately modifies ℓ_1's *rin* variable, effectively marking itself entitled, and waits at Line 12 for $\mathcal{R}_2^{w,nn}$ to complete. When $\mathcal{R}_4^{w,nn}$ is issued, it must wait at Line 15 (Listing 4) for $\mathcal{R}_3^{r,n}$ to complete, even though it should have been immediately satisfied following the rules of the fast RW-RNLP.

Using the implementation given in Listing 5, however, $\mathcal{R}_3^{r,n}$ must wait at Line 6 due to $\mathcal{R}_2^{w,nn}$ having marked itself present in the bottom byte of ℓ_1's *rin* variable. This prevents $\mathcal{R}_3^{r,n}$ from modifying ℓ_1's *rin* variable before it becomes entitled. Therefore, when $\mathcal{R}_4^{w,nn}$ is issued, the condition at Line 15 in Listing 4 is true for its single resource ℓ_2, so it is immediately satisfied.

3.4.3 Linearizability

Herlihy and Wing presented linearizability as a correctness condition for concurrent objects that "provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point

between its invocation and its response." Linearizability is a local property; if the operations on each object can be linearized, the system as a whole is considered to be linearizable (Herlihy and Wing, 1990). In the following discussion, the fast RW-RNLP with the RW-RNLP* is used as the example, but the fast RW-RNLP with the R^{3}LP is linearizable as well.

Section 3.2.6 claims that each routine presented has a linearization point; this is the point at which the routine can be considered to take effect (atomically). For the non-nested routines, these points are clear. A read request enqueues atomically at Line 3 (Listing 4) and can be viewed as executing the lock function as a whole atomically at the end of the procedure. Similarly for R*_UNLOCK^{nn}, the routine's linearization point can be considered to be at its invocation. The non-nested write routines function similarly, with linearization points at the end of the lock routine's execution and the beginning of the execution of the unlock routine.

The nested routines grant access to groups of resources at a time (Listing 5). Considering the routines themselves, each call of the lock routine can be said to linearize to the last point in its execution. That is, no access to any of the requested resources occurs before that point in time, and the order of request accesses to those resources is exactly the order of termination of the lock routines. (Recall that linearization is defined relative to a specific resource; there may be requests for other resources occurring concurrently. These requests are not granted access clearly before or after the non-conflicting request. Again, linearization is a local property and there may be multiple legal sequential histories (Herlihy and Wing, 1990).) Just like non-nested requests, the invocation of each unlock routine can be considered to be the linearization point of the entire routine.

An example of the linearization of several objects is shown in Figure 3.18. An operation invocation *op* on a set of shared resources D_i by request $\mathcal{R}_i$ is indicated by D_i *op* $\mathcal{R}_i$ above a line whose length corresponds to the duration of time each invocation takes. The linearization point of each operation's execution is indicated with a circle at some point during its execution. As discussed above, this point can always be selected at the end of the execution of a `lock` operation and at the beginning of the execution of an `unlock` operation. In Figure 3.18, time moves forward to the right.

Example 3.11. In Figure 3.18, $\mathcal{R}_1^{w,n}$ is the first to begin executing the lock logic to gain mutually exclusive access to $D_1 = \{\ell_1, \ell_2\}$. Then, $\mathcal{R}_2^{w,nn}$ is issued for $D_2 = \{\ell_2\}$. $\mathcal{R}_2^{w,nn}$ calls W*_LOCK^{nn} for ℓ_2. It is granted access to ℓ_2 first (at the end of the lock routine), and then enters its critical section before calling W*_UNLOCK^{nn}. During $\mathcal{R}_2^{w,nn}$'s execution of the lock operation, $\mathcal{R}_3^{r,n}$ invoked the lock call for $D_3 = \{\ell_1, \ell_2\}$.

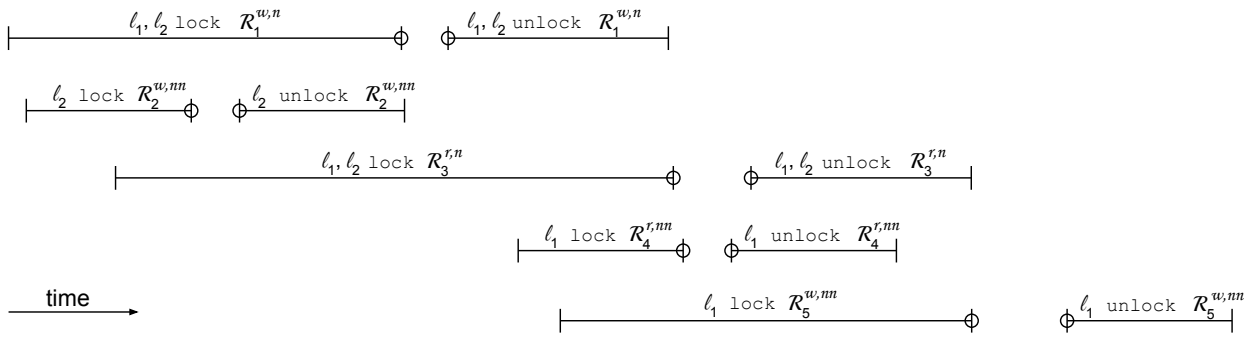


Figure 3.18: Illustration of a series of lock and unlock calls by requests $\mathcal{R}_1$ through $\mathcal{R}_5$ with the linearization point of each operation shown with a circle.

At some point $\mathcal{R}_2^{w,nn}$ completes its critical section and invokes the unlock routine. The unlock routine can be linearized to the point indicated in the Figure 3.18, which clearly comes before the point at which $\mathcal{R}_1^{w,n}$ or $\mathcal{R}_3^{r,n}$ has linearized its respective lock call. Note that this properly reflects the mutually exclusive access for $\mathcal{R}_2^{w,nn}$ for ℓ_2; a request is considered to access the resource between the linearization point for its call to the lock routine and the linearization point for its call to the unlock routine.

At a later point in time, $\mathcal{R}_1^{w,n}$ finishes execution of the lock routine, enters its critical section, and then calls the unlock routine.

While $\mathcal{R}_1^{w,n}$ is executing the unlock routine for $D_1 = \{\ell_1, \ell_2\}$, $\mathcal{R}_4^{r,nn}$ and $\mathcal{R}_5^{w,nn}$ are issued for $D_4 = \{\ell_1\}$ and $D_5 = \{\ell_1\}$, respectively.

At some point in time after $\mathcal{R}_1^{w,n}$ has updated the writer bits of ℓ_1's *rin* variable, $\mathcal{R}_4^{r,nn}$ becomes satisfied and completes its call to the lock routine. Similarly, after $\mathcal{R}_1^{w,n}$ has updated ℓ_2's *rin* variable, $\mathcal{R}_3^{r,n}$ becomes satisfied and completes its call to the lock routine. Note that overlapping critical sections for ℓ_1 is expected behavior for these requests; read requests may overlap.

Once the read requests finish accessing their respective resources, they both call the unlock routine. At a future point in time, $\mathcal{R}_5^{w,nn}$ completes its call to the lock routine and can begin its critical section. Note that the linearization points correctly reflect mutually exclusive access for this request for ℓ_1. $\Diamond$

3.4.4 Constraints used in Schedulability Study

As mentioned in Section 3.3.2, the calculated blocking from the worst-case acquisition-delay bounds presented in Table 3.1 is refined by applying several constraints. These are illustrated below with an example using a ticket lock, followed by a discussion of how these can be applied to more complex protocols.

Example 3.12. Consider an application with four tasks ($n = 4$) and six processors ($m = 6$) in which all tasks have the same period and access the same resource. The access of this resource is protected by a ticket lock. The requests issued by these tasks are $\mathcal{R}_1^w, ..., \mathcal{R}_4^w$, with critical-section lengths $L_1 = 50\mu$s, $L_2 = 60\mu$s, $L_3 = 30\mu$s, and $L_4 = 20\mu$s. When calculating the worst-case blocking that the task issuing request $\mathcal{R}_1^w$ may experience, the analytical bound for a ticket lock of $(m-1)L_{max}^w$, where $L_{max}^w = 60\mu$s can be used; the blocking for $\mathcal{R}_1^w$ is bounded by $5 \cdot 60 = 300\mu$s. However, this blocking can never occur in practice. ◊

Period-based constraints. Based on the period of each task in the system, the number of jobs (and thus requests) of each task that may be active while a given request is active can be computed. The analysis can then be tightened by selecting only the longest instances of critical sections that may delay a given request.

Example 3.12 (continued). As all tasks have the same period, at most two jobs of a given task can overlap with the job of interest. Thus, the job that issues $\mathcal{R}_2^w$ can only issue it twice while $\mathcal{R}_1^w$ is waiting or executing. The analysis can be tightened to select the top $m-1$ instances of critical sections that may delay $\mathcal{R}_1^w$. The new bound on the worst-case blocking after applying period-based constraints is $60+60+30+30+20 = 200\mu$s. Note that $\mathcal{R}_1^w$ is not counted as potentially increasing its own blocking. ◊

FIFO constraints. The write requests handled by a ticket lock are enqueued in a FIFO queue. Thus, each request can only delay a given request once.

Example 3.12 (continued). By imposing FIFO constraints, worst-case blocking can be bounded to $60+30+20 = 110\mu$s. ◊

Example 3.13. For the next scenario, the task system presented in Example 3.12 is modified by adding a read request $\mathcal{R}_5^r$ with $L_5 = 3\mu$s. The task that issues $\mathcal{R}_5^r$ has a shorter period such that it could be issued six times while a given write request is active. Instead of using a ticket lock, a PF-TL is applied here. When considering the blocking $\mathcal{R}_1^w$ may experience, the above analysis still holds for the write requests. Now the blocking $\mathcal{R}_1^w$ may experience due to read requests must be considered. ◊

Note that when considering locking protocols with both read and write requests, FIFO constraints may apply to the write requests but will not apply to read requests; all reader/writer protocols considered here handle read requests separately from write requests to yield constant-time blocking. Because of this, a given read request may execute multiple times before a write request of interest. However, the total number of read requests that must be included can be constrained by period-based constraints and a new constraint.

Read-write constraints. Each protocol examined in this chapter functions by alternative write and read phases in some manner. Thus, the number of *read* phases by which a request is blocked is constrained by the number of *write* phases possible, which may be limited by the above constraints.

Example 3.13 (continued). In this scenario, only three write requests could contribute to the blocking $\mathcal{R}_1^w$ experiences. In the worst case, before each of these write requests (including $\mathcal{R}_1^w$), a read phase could occur. This limits the worst case to including the four longest read critical-section instances in the blocking. Because the read request $\mathcal{R}_5^r$ could occur six times, its critical section can be counted for all four instances. Thus, the blocking $\mathcal{R}_1^w$ experiences due to read requests is at most $4 \cdot 3 = 12\mu$s and the total blocking is at most $110 + 12 = 122\mu$s. $\Diamond$

While the above examples have focused on the blocking a write request may experience, the same constraints may be applied when calculating the write and read critical sections that may block a read request.

Contention constraints. For both fast RW-RNLP variants, the blocking bound for non-nested write requests depends on contention. Recall that in Theorem 3.1, the contending requests that contributed to blocking were only other non-nested write requests for the same resource. Therefore, for each task system and each request, the number of contending non-nested write requests is used for the value of contention.

Constraints applied to protocols. Recall that four protocols were evaluated: the PF-TL, the RW-RNLP, the fast RW-RNLP with the R^{3}LP, and the fast RW-RNLP with the RW-RNLP*. The PF-TL was applied as group lock for a static group of all resources.

When the period-based, FIFO, and read-write constraints are applied to the group PF-TL, the blocking computed for read and write requests may be tightened. The same analysis holds for the RW-RNLP. While the RW-RNLP is a fine-grained locking protocol, its worst-case blocking for write requests is still $(m-1)(L_{max}^w + L_{max}^r)$ due to potential transitive blocking chains between nested write requests. Computing these transitive blocking chains is an expensive process, and in a system with randomly requested resources, write expansion and potential chains imply that it is likely that each write request could delay any other write request. However, the FIFO manner in which the RW-RNLP functions allows us to use the FIFO constraint to limit the contribution of each write request to blocking others only once. Without expensive tighter analysis that considers possible transitive blocking chains, the tightened blocking bounds of the RW-RNLP are identical to those of the group PF-TL.

FIFO constraints can also be applied to the fast RW-RNLP, but the modular structure means that FIFO constraints cannot be applied to write requests of the opposite type. For example, a given nested write request may enter the RNLP and then be allowed to execute by the global arbitration mechanism multiple times while a non-nested write request waits behind other non-nested write requests in its ticket lock. The same scenario can occur for nested write requests. Thus, the FIFO constraints can only be applied to nested write requests which share at least one resource or non-nested write requests for the same resource. For any write requests to which FIFO constraints cannot be applied, the period-based constraints can still be applied. Due to the way in which requests are grouped by the R^3LP, when accounting for blocking in the fast RW-RNLP with the R^3LP, the highest critical-section length instances of non-nested write requests and nested write requests are added separately to get a tighter bound.

Based on the number of requests being generated in each scenario, it is possible that in some scenarios, the manner in which the FIFO constraints can be applied in the context of the group PF-TL may greatly reduce the number of critical sections that may be counted toward blocking. (Because it is a group lock, each write can only effect the request of interest once by the FIFO constraint.) The FIFO constraint cannot be applied as broadly to the fine-grained nesting protocols, as some requests may delay the request of interest multiple times. This argues for tighter blocking analysis, though getting exact blocking analysis for nested resource accesses is NP-hard (Wieder and Brandenburg, 2014).

3.5 Chapter Summary

This chapter has presented a new RNLP variant called the fast RW-RNLP. The fast RW-RNLP employs a fast-path mechanism to provide contention-sensitive pi-blocking and low processing costs for read and non-nested write requests requests, while preserving the RW-RNLP's asymptotic pi-blocking bounds for nested requests. This provides a solution for the common case while also supporting the less common nested write requests. In the experimental evaluation, lock/unlock overhead for non-nested requests is nearly identical under the fast RW-RNLP and PF-TLs. Additionally, observed pi-blocking times for such requests are reduced compared to the RW-RNLP. This is because non-nested requests require less overhead and are immune to transitive blocking effects under the fast RW-RNLP. These results for overhead and pi-blocking are reflected in the large-scale schedulability study, in which the fast RW-RNLP variants, and the R^3LP in

particular, tended to result in higher schedulability in scenarios in which non-nested resource access was the common case.

Acknowledgment. The work presented in this chapter first appeared in papers written by Tanya Amert and myself. Tanya was involved throughout the project, and in particular, she implemented both protocol variants and led the evaluation.

CHAPTER 4: MINIMIZING IMPACTS ON NESTED WRITE REQUESTS[1]

The previous chapter presented approaches to mitigate the effect of transitive blocking chains on read requests and on non-nested write requests. The focus of this chapter is on handling nested write requests.

As described in Chapter 2, existing RNLP variants yield asymptotically optimal blocking when n and m are considered as variables. However, while the RNLP family of protocols grants fine-grained resource access, the $O(m)$ blocking for the non-preemptive spin-based variant is no better than the blocking under a coarse-grained locking approach. This motivates the consideration of resource contention in both protocol analysis and protocol development.

This chapter presents a new RNLP variant, the contention-sensitive RNLP (C-RNLP). The key idea behind the C-RNLP is to allow requests to safely "cut ahead" of previously issued requests to yield contention-sensitive blocking. This is accomplished by integrating critical-section lengths into the ordering logic of the protocol. As critical-section lengths are required for schedulability analysis, their availability is not an unreasonable assumption for real-time systems.

The C-RNLP is presented first as a set of rules that govern requests along with bounds on blocking (Section 4.1). Then, details on its implementation are given (Section 4.2), followed by an experimental evaluation of the overhead and blocking of the C-RNLP (Section 4.3).

The C-RNLP has high overhead, which motivates the development of an another protocol (Section 4.4). This protocol, the *Concurrency Group Locking Protocol (CGLP)*, achieves low overhead. Though the CGLP is not strictly contention sensitive, depending on the system, it can significantly reduce the previous state-of-the-art $O(m)$ blocking. The CGLP is specified by both an offline and an online component, along

[1]Contents of this chapter previously appeared in preliminary form in the following papers:

Jarrett, C., Ward, B., and Anderson, J. (2015). A contention-sensitive fine-grained locking protocol for multiprocessor real-time systems. In *Proceedings of the 23rd International Conference on Real-Time Networks and Systems*.

Nemitz, C., Amert, T., Goyal, M., and Anderson, J. (2019b). Concurrency groups: A new way to look at real-time multiprocessor lock nesting. In *Proceedings of the 27th International Conference on Real-Time Networks and Systems*.

Nemitz, C., Amert, T., Goyal, M., and Anderson, J. (2021a). Concurrency groups: A new way to look at real-time multiprocessor lock nesting. *Real-Time Systems*, 57(1):190–226. Reproduced with permission from Springer Nature.

with bounds on blocking (Section 4.5). Extensions to the protocols are then presented (Sections 4.6–4.8). Next, the CGLP is evaluated on the basis of overhead (Section 4.9).

Finally, both the C-RNLP and the CGLP are evaluated on the basis of schedulability with a large-scale study (Section 4.10).

4.1 C-RNLP

The C-RNLP is the first fine-grained locking protocol with contention-sensitive worst-case blocking for nested requests. This is accomplished by allowing some lock requests to "cut ahead" of other queued requests, thereby limiting the length of transitive blocking chains. This enables blocking under the C-RNLP to be $O(\min(m, N_i))$, where m is the number of processors and N_i is the contention of $\mathcal{R}_i$.

The C-RNLP is defined in two passes. First, an abstract description shows how the wait-for graph representing active requests is updated as requests are issued and completed. Second, in Section 4.2, a more concrete implementation that conforms to the abstract specification is described.

The ordering of requests is maintained in a directed, acyclic wait-for graph $G = (V, E)$, where vertices denote requests and edges denote waiting relationships, *i.e.*, $(\mathcal{R}_i, \mathcal{R}_j) \in E$ means that $\mathcal{R}_i$ is blocked by $\mathcal{R}_j$. Initially, G is empty, with $V = \emptyset$ and $E = \emptyset$. When a request is issued, it is added to the graph. This addition of a request along with any associated edges is called an *insertion*. Likewise, when a request completes, the removal of it and all of its edges is called a *removal*. The graph that results from G after an insertion or a removal is denoted $G' = (V', E')$. (Similarly, primes are used in referring to notation relevant to G'.) The graph G' is *instantiated* when it results from applying an insertion or removal operation on G. For now, let us assume that these operations are atomic and take zero time to apply. $\mathcal{R}_i$ is *satisfied* when it has no outgoing edges. A resource ℓ_a requested by $\mathcal{R}_i$ is *locked* by $\mathcal{R}_i$ when $\mathcal{R}_i$ is satisfied. A protocol is considered *safe* if at most one request can lock any one resource at a time. Once $\mathcal{R}_i$ is satisfied, $\mathcal{R}_i$ completes within L_i time units. Later, the implications of the violation of this assumption are explored. For now, a series of examples is used to motivate the rules of the C-RNLP.

4.1.1 Safety

This first example motivates the rules presented later that ensure that the C-RNLP is safe.

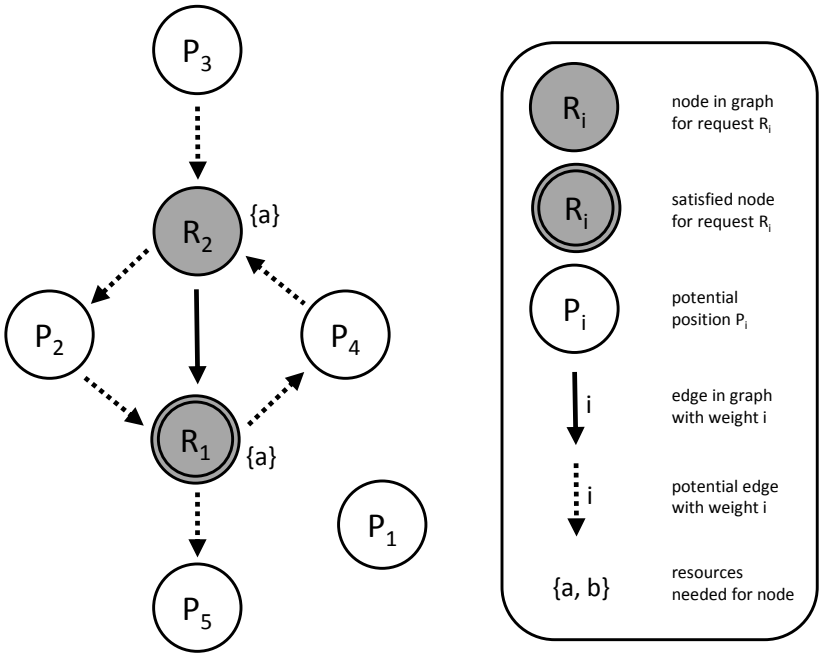


Figure 4.1: A wait-for graph G and several positions at which $\mathcal{R}_3$ could be inserted. (The legend also applies to subsequent figures. Note that edge weights are not used in this particular figure.)

Example 4.1. Consider the wait-for graph G shown in Figure 4.1. Each request requires only ℓ_a. $\mathcal{R}_1$ is satisfied and holds ℓ_a and blocks $\mathcal{R}_2$, as shown by the directed arrow to $\mathcal{R}_1$. Now suppose that $\mathcal{R}_3$ is issued and requires ℓ_a. $\mathcal{R}_3$ is added to G, and we must consider which edges to add. Several positions for inserting $\mathcal{R}_3$ are displayed in Figure 4.1, denoted as positions P_1–P_5. For now, it suffices to understand the notion of a position intuitively, but later a formal definition is needed. Intuitively, when a request is inserted into G, it is implicitly *reserving* a position. (Actually, with DGLs, a *set* of positions is reserved, but this additional complication is ignored for now.) After presenting several definitions, these positions are examined. $\Diamond$

Definition 4.1. For any request $\mathcal{R}_i$, let $In(\mathcal{R}_i)$ denote its incoming edges, $In(\mathcal{R}_i) = \{(\mathcal{R}_j, \mathcal{R}_i) : (\mathcal{R}_j, \mathcal{R}_i) \in E\}$, and let $Out(\mathcal{R}_i)$ denote its outgoing edges, $Out(\mathcal{R}_i) = \{(\mathcal{R}_i, \mathcal{R}_j) : (\mathcal{R}_i, \mathcal{R}_j) \in E\}$.

Definition 4.2. Let S be the set of satisfied requests: $S = \{\mathcal{R}_i : Out(\mathcal{R}_i) = \emptyset\}$.

Definition 4.3. A request $\mathcal{R}_i$ *precedes* $\mathcal{R}_j$, denoted $\mathcal{R}_i \prec \mathcal{R}_j$, if there exists a directed path from $\mathcal{R}_j$ to $\mathcal{R}_i$.

Definition 4.4. A request $\mathcal{R}_i$ has *cut ahead* of $\mathcal{R}_j \in V$ if G' is obtained by the insertion of $\mathcal{R}_i$ and $\mathcal{R}_i \prec \mathcal{R}_j$ is established.

Example 4.1 (continued). Let us examine each of the positions P_1–P_5:

- P_1. Here $\mathcal{R}_3$ would have no edges, and thus would be satisfied. However, this would lead to ℓ_a being locked by both $\mathcal{R}_1$ and $\mathcal{R}_3$, which violates safety. Therefore, the protocol should not allow $\mathcal{R}_3$ to be placed at P_1.

- P_2. Here $\mathcal{R}_3$ would cut ahead of $\mathcal{R}_2$ and would be waiting for $\mathcal{R}_1$ to finish. This position ensures safety, *i.e.*, ℓ_a will be locked by at most one request at a time.

- P_3. Here $\mathcal{R}_3$ would not cut ahead of any request, and would be waiting for $\mathcal{R}_2$. This position is also safe.

- P_4. Here $\mathcal{R}_3$ would cut ahead of $\mathcal{R}_1$ and wait for $\mathcal{R}_2$. This maintains safety, but creates deadlock in the system.

- P_5. Here $\mathcal{R}_3$ would be cutting ahead of $\mathcal{R}_1$, which should be disallowed since $\mathcal{R}_1$ is already satisfied.

$\Diamond$

Motivated by the above example, note that to ensure safety, there must be a single order in which each request for the same resource will be processed. A newly inserted request should also avoid cutting ahead of a request that has already been satisfied.

Definition 4.5. A set of requests $Q \subseteq V$ has a *unique ordering* if and only if for any two distinct requests $\mathcal{R}_i$ and $\mathcal{R}_j$ in Q, either $\mathcal{R}_i \prec \mathcal{R}_j$ or $\mathcal{R}_j \prec \mathcal{R}_i$.

Definition 4.6. Let Q_a be the set of requests that require the resource ℓ_a: $Q_a = \{\mathcal{R}_i : \mathcal{R}_i \in V \wedge \ell_a \in \mathcal{D}_i\}$.

Definition 4.7. An insertion into G resulting in G' is a *safe insertion* if: **(i)** for each resource ℓ_a, there is a unique ordering on the set Q'_a; and **(ii)** a new request $\mathcal{R}_i$ does not cut ahead of a satisfied request, *i.e.*, for any $\mathcal{R}_j$ in V, $(\mathcal{R}_j, \mathcal{R}_i) \in E' \Rightarrow \mathcal{R}_j \notin S$. (Note that S is the set of satisfied requests in G.)

4.1.2 Delay Preservation

Now that we have determined which insertions are safe, we investigate which are "best." In order to do so, information is added to G about how long each request will run.

Definition 4.8. The *weight* of an edge $(\mathcal{R}_i, \mathcal{R}_j) \in E$ is given by $W(\mathcal{R}_i, \mathcal{R}_j) = L_j$.

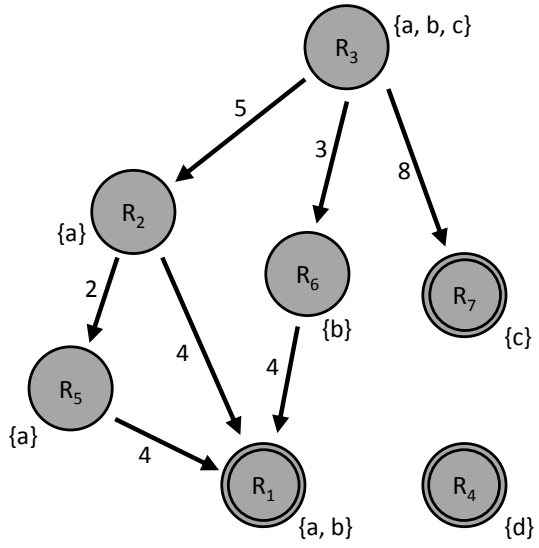


Figure 4.2: A wait-for graph G that includes seven requests.

Example 4.2. Suppose G starts by containing nodes $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_3$ and the edges depicted between them as shown in Figure 4.2, after which, $\mathcal{R}_4$, $\mathcal{R}_5$, $\mathcal{R}_6$, and $\mathcal{R}_7$ are inserted in order into G. Let us examine each of those insertions.

- $\mathcal{R}_4$. The insertion of this request with no edges is clearly safe, as no other request in G requires ℓ_d.
- $\mathcal{R}_5$. This request cuts ahead of $\mathcal{R}_2$. While this is a safe insertion, it increases $\mathcal{R}_2$'s blocking time, as $\mathcal{R}_2$ must now wait for up to $L_1 + L_5 = 6$ time units to execute, as opposed to waiting for at most $L_1 = 4$ time units.
- $\mathcal{R}_6$. This request cuts ahead of $\mathcal{R}_3$ and waits for $\mathcal{R}_1$ to complete. Since $\mathcal{R}_3$ is already waiting for as much as $L_1 + L_2 = 9$ time units, $\mathcal{R}_6$ cutting ahead is acceptable, as it would cause at most $L_1 + L_6 = 7$ time units of waiting time.
- $\mathcal{R}_7$. This request cuts ahead of $\mathcal{R}_3$. Using the same reasoning as used with $\mathcal{R}_6$, note that $L_7 < L_1 + L_2$. However, since $L_7 > L_2$, if $\mathcal{R}_1$ had already been running for close to L_1 time units, then inserting $\mathcal{R}_7$ in this position could delay $\mathcal{R}_3$.

$\Diamond$

As demonstrated in Example 4.2, it is necessary to know how long a satisfied request has been satisfied in order to reason about where to insert a request.

Definition 4.9. The *running time* of a request $\mathcal{R}_i$, denoted r_i, is the time for which $\mathcal{R}_i$ has been satisfied.

Note that under the current assumption that $\mathcal{R}_i$ completes within L_i time units, $r_i < L_i$.

Definition 4.10. A *path* in G from $\mathcal{R}_i \in V$ to $\mathcal{R}_j \in V$ is denoted $\mathcal{R}_i \rightsquigarrow \mathcal{R}_j$. The length of this path is given by the sum of the weights of the included edges. This is denoted $|\mathcal{R}_i \rightsquigarrow \mathcal{R}_j|$.

As seen in Example 4.2, determining a request $\mathcal{R}_i$'s maximum blocking time requires the maximum distance traversed through G from $\mathcal{R}_i$'s node to a satisfied node, and must take into account r_j for any $\mathcal{R}_j$ that is satisfied. $\mathcal{R}_i$'s maximum blocking time is given by the value $|A(\mathcal{R}_i)|$, defined next.

Definition 4.11. If $\mathcal{R}_i \notin S$, then let $A(\mathcal{R}_i)$ denote a path $\mathcal{R}_i \rightsquigarrow \mathcal{R}_j$ such that $\mathcal{R}_j \in S$ and $|\mathcal{R}_i \rightsquigarrow \mathcal{R}_j| - r_j$ is maximal; in this case, let us define $|A(\mathcal{R}_i)| = |\mathcal{R}_i \rightsquigarrow \mathcal{R}_j| - r_j$. If $\mathcal{R}_i \in S$, then let us define $|A(\mathcal{R}_i)| = 0$.

As suggested by Example 4.2, the goal for this protocol is to preserve the existing maximum delays for each request. Requiring preservation of maximum delays would also prevent the possibility of deadlock as shown in Example 4.1.

Definition 4.12. An insertion into G is *delay-preserving* if and only if $(\forall \mathcal{R}_i \in V :: |A'(\mathcal{R}_i)| \leq |A(\mathcal{R}_i)|)$.

4.1.3 C-RNLP Rules

Motivated by the examples above, the C-RNLP implementation must satisfy the following rules:

Rule 1: All requests wait until satisfied.

Rule 2: $\mathcal{R}_i$ is removed when $\mathcal{R}_i$ completes.

Rule 3: A node is inserted at a safe, delay-preserving position in G that gives the lowest $|A(R_i)|$.

Rule 4: Insertions into and removals from G are atomic.

Using the rules of the C-RNLP, sometimes results in inserting requests into G in different positions than if the RNLP were in use; the RNLP orders requests in the order they are issued, with no cutting ahead.

Refining Rule 3. The abstract rules that define the C-RNLP have been presented, but Rule 3 lacks sufficient information to guide an actual implementation. Therefore, Rule 3 will be refined after giving some necessary definitions. First, the notion of a position is refined.

Definition 4.13. $\mathcal{R}_i$ and $\mathcal{R}_j$ are *consecutive with respect to* ℓ_a if $\{\mathcal{R}_i, \mathcal{R}_j\} \subseteq Q_a \wedge \mathcal{R}_i \prec \mathcal{R}_j \wedge \neg(\exists \mathcal{R}_l : \mathcal{R}_l \in Q_a :: \mathcal{R}_i \prec \mathcal{R}_l \prec \mathcal{R}_j)$.

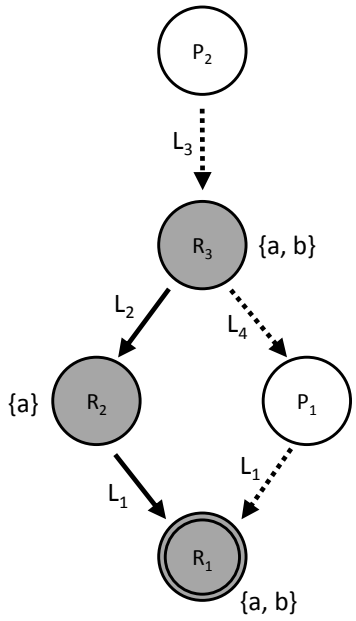


Figure 4.3: A wait-for graph G with two possible positions for $\mathcal{R}_4$.

Example 4.3. Given the graph shown in Figure 4.3, $\mathcal{R}_1$ and $\mathcal{R}_2$ are consecutive with respect to ℓ_a, and $\mathcal{R}_1$ and $\mathcal{R}_3$ are consecutive with respect to ℓ_b. ◊

Definition 4.14. A *position* P_k has at most one incoming edge and at most one outgoing edge, which in an abuse of previous notation (Definition 4.1), are denoted $In(P_k)$ and $Out(P_k)$, respectively.

- If $In(P_k) = \emptyset$, then P_k is called a *top-most* position.
- If $Out(P_k) = \emptyset$, then P_k is called a *bottom-most* position.
- Otherwise, if $In(P_k) = \mathcal{R}_i$ and $Out(P_k) = \mathcal{R}_j$, then P_k is called an *inner* position. An inner position must have $Out(P_k) \prec In(P_k)$.

A position is said to be an ℓ_a-*position* if $In(P_k)$ and $Out(P_k)$ are each either $\emptyset$ or a request that includes ℓ_a.

When a request $\mathcal{R}_l$ is inserted into a graph G, it *reserves* a set of positions X, and $In'(\mathcal{R}_l) = \bigcup_{P_k \in X} In(P_k)$ and $Out'(\mathcal{R}_l) = \bigcup_{P_k \in X} Out(P_k)$.

Example 4.3 now motivates the discussion of a position's capacity, defined below.

Example 4.3 (continued). Suppose G is as shown in Figure 4.3, with request $\mathcal{R}_4$ for resource ℓ_b about to be added to the graph at either position P_1 or position P_2. Both positions would yield a safe insertion. Whether $\mathcal{R}_4$'s insertion at P_1 is delay-preserving depends on the values of L_2 and L_4 and how much longer $\mathcal{R}_1$ will be

executing in the worst case. For this insertion to be delay-preserving, $\mathcal{R}_4$ must finish at the latest when $\mathcal{R}_2$ would finish in the worst case (where each request takes exactly its stated maximum time to execute), as this would ensure that all later requests—only $\mathcal{R}_3$ in this scenario—would experience no additional worst-case blocking. The concept of position *capacity* is introduced in order to reason about what values of L_4 would meet this condition. ◊

Definition 4.15. The *capacity* of a position P_k is defined as:

$$cap(P_k) = \begin{cases} \infty & \text{if } In(P_k) = \emptyset \\ |A(\mathcal{R}_i)| & \text{if } In(P_k) = \mathcal{R}_i \wedge Out(P_k) = \emptyset \\ \omega - \beta & \text{otherwise} \end{cases} \tag{4.1}$$

where $\omega = |A(\mathcal{R}_i)| - |A(\mathcal{R}_j)|$ and $\beta = (L_j - r_j)$ for $In(P_k) = \mathcal{R}_i \wedge Out(P_k) = \mathcal{R}_j$.

Example 4.3 (continued). We can now reason about the capacities of P_1 and P_2. For P_1, the capacity computation yields $cap(P_1) = |A(\mathcal{R}_3)| - |A(\mathcal{R}_1)| - (L_1 - r_1) = (L_1 + L_2 - r_1) - 0 - L_1 + r_1 = L_2$. This value indicates how long $\mathcal{R}_3$ will be waiting in the worst case due solely to its blocking on $\mathcal{R}_2$ and taking into account that any request reserving position P_1 would also be waiting for $\mathcal{R}_1$ to finish. Therefore, if L_4 were at most this value, $\mathcal{R}_4$ could be inserted into P_1 and be delay-preserving. Because $In(P_2) = \emptyset$, $cap(P_2) = \infty$. Intuitively, a request of any length could be inserted into P_2. ◊

Definition 4.16. Let $P_{\mathcal{D}_i}$ be a $\mathcal{D}_i$ *position set*, such that for each $\ell_a \in \mathcal{D}_i$ there is an ℓ_a-position in $P_{\mathcal{D}_i}$.

When a request $\mathcal{R}_i$ is issued, it must reserve all positions in a $\mathcal{D}_i$ position set, $P_{\mathcal{D}_i}$. In turn, $\mathcal{R}_i$ is inserted into G, with $In(\mathcal{R}_i) = \bigcup_{P_k \in P_{\mathcal{D}_i}} \{In(P_k)\}$ and $Out(\mathcal{R}_i) = \bigcup_{P_k \in P_{\mathcal{D}_i}} \{Out(P_k)\}$.

Definition 4.17. The *capacity* of the position set $P_{\mathcal{D}_i}$ is the smallest $|A(\mathcal{R}_i)|$ where $\mathcal{R}_i \in In(P_{\mathcal{D}_i})$ (or ∞ if $In(P_{\mathcal{D}_i}) = \emptyset$) minus the largest $|A(\mathcal{R}_j)| + L_j$ where $\mathcal{R}_j \in Out(P_{\mathcal{D}_i})$ (or 0 if $Out(P_{\mathcal{D}_i}) = \emptyset$)

Note that the capacity of a position set is at most that of each individual position in the set.

Now that the concepts pertaining to positions are more fully developed, Rule 3 is replaced with Rule 3′, which upholds Rule 3 and refines how safe and delay-preserving positions are found.

Rule 3′: $\mathcal{R}_i$ is inserted by reserving all positions in $P_{\mathcal{D}_i}$ where $L_i \leq cap(P_{\mathcal{D}_i})$ and $|A(\mathcal{R}_i)|$ is minimized.

Example 4.4. Suppose G is as shown in Figure 4.4. (Note that all possible positions are shown.) Now that Rule 3′ is defined, it is possible to easily examine all possible positions that a request $\mathcal{R}_i$ may reserve and

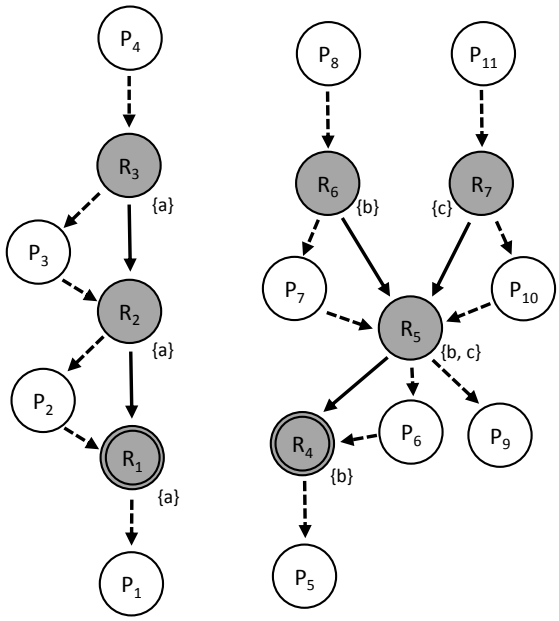


Figure 4.4: A wait-for graph G with all possible positions shown. P_1–P_4 are ℓ_a-positions, P_5–P_8 are ℓ_b-positions, and P_9–P_{11} are ℓ_c-positions.

determine which minimizes $|A(\mathcal{R}_i)|$. If $\mathcal{D}_i = \{\ell_a\}$, then $\mathcal{R}_i$ must reserve one of the positions P_1–P_4. If instead $\mathcal{D}_i = \{\ell_a, \ell_b\}$, then $\mathcal{R}_i$ must reserve both an ℓ_a position P_1–P_4 and an ℓ_b position P_5–P_8. Thus, $\mathcal{R}_i$ will have at least two edges. ◊

The fact that Rule 3′ upholds Rule 3 follows from the next three lemmas, the latter two of which are stated without proof, as they are straightforward.

Lemma 4.1. *$\mathcal{R}_i$ inserted under Rule 3′ is delay-preserving.*

Proof. Suppose to the contrary that $\mathcal{R}_i$ reserves a set of positions $P_{\mathcal{D}_i}$ and the insertion of $\mathcal{R}_i$ is not delay-preserving. Then, there is at least one node that obtains a new edge directed to $\mathcal{R}_i$ that experiences increased blocking. Let $\mathcal{R}_j$ denote such a node. Because $\mathcal{R}_j$'s blocking increases,

$$|A'(\mathcal{R}_j)| > |A(\mathcal{R}_j)|. \tag{4.2}$$

Because the insertion of $\mathcal{R}_i$ caused $\mathcal{R}_j$'s blocking to increase, $|A'(\mathcal{R}_j)|$ depends on $\mathcal{R}_i$, which immediately upon insertion has not yet had any running time (and may in fact be waiting on another request). Therefore, by Definition 4.11,

$$|A'(\mathcal{R}_j)| = |A'(\mathcal{R}_i)| + L_i. \tag{4.3}$$

Let $\mathcal{R}_l$ denote a node to which an outgoing edge from $\mathcal{R}_i$ is directed due to the insertion such that the value $|A(\mathcal{R}_l)| + L_l$ is maximized. If no such node $\mathcal{R}_l$ exists, then the following establishes $cap(P_{\mathcal{D}_i}) < L_i$.

$$\begin{aligned}
cap(P_{\mathcal{D}_i}) &\leq \{\text{by Definition 4.17}\} \\
&\quad |A(\mathcal{R}_j)| - 0 \\
&< \{\text{by (4.2)}\} \\
&\quad |A'(\mathcal{R}_j)| \\
&= \{\text{by (4.3)}\} \\
&\quad |A'(\mathcal{R}_i)| + L_i \\
&= \{\text{by Definition 4.11, as } \mathcal{R}_i \in S\} \\
&\quad L_i
\end{aligned}$$

If instead there does exist an $\mathcal{R}_l$, then, by Definition 4.11,

$$|A'(\mathcal{R}_i)| = |A'(\mathcal{R}_l)| + L_l - r_l. \tag{4.4}$$

Note that, after the insertion of $\mathcal{R}_i$, there can be no path from $\mathcal{R}_l$ to $\mathcal{R}_i$, for then a cycle would exist (and hence, deadlock). More technically, if such a cycle were caused, then the set of positions $P_{\mathcal{D}_i}$ would have zero or negative capacity by Definition 4.17, and thus $\mathcal{R}_i$ would not have been inserted by Rule 3′. Because the insertion of $\mathcal{R}_i$ does not result in any path from $\mathcal{R}_l$ to $\mathcal{R}_i$,

$$|A'(\mathcal{R}_l)| = |A(\mathcal{R}_l)|. \tag{4.5}$$

The following reasoning establishes $cap(P_{\mathcal{D}_i}) < L_i$.

$$\begin{aligned}
cap(P_{\mathcal{D}_i}) &\leq \{\text{by Definition 4.17}\} \\
&\quad |A(\mathcal{R}_j)| - (|A(\mathcal{R}_l)| + L_l) \\
&< \{\text{by (4.2)}\} \\
&\quad |A'(\mathcal{R}_j)| - (|A(\mathcal{R}_l)| + L_l) \\
&= \{\text{by (4.5)}\} \\
&\quad |A'(\mathcal{R}_j)| - (|A'(\mathcal{R}_l)| + L_l) \\
&= \{\text{by (4.3)}\} \\
&\quad |A'(\mathcal{R}_i)| + L_i - (|A'(\mathcal{R}_l)| + L_l)
\end{aligned}$$

$$
\begin{aligned}
&= \{\text{by (4.4)}\} \\
&\quad |A'(\mathcal{R}_l)| + L_l - r_l + L_i - (|A'(\mathcal{R}_l)| + L_l) \\
&= \{\text{by simplification}\} \\
&\quad L_i - r_l \\
&\leq \{\text{because } r_l \geq 0\} \\
&\quad L_i
\end{aligned}
$$

Therefore, for $\mathcal{R}_j$ to have experienced increased blocking upon the insertion of $\mathcal{R}_i$, $\mathcal{R}_i$ must have reserved a set of positions with $cap(P_{\mathcal{D}_i}) < L_i$, which violates Rule 3′. Thus, such a set of reservations cannot occur. □

Lemma 4.2. *$\mathcal{R}_i$ inserted under Rule 3′ is safe.*

Lemma 4.3. *$\mathcal{R}_i$ is inserted by following Rule 3′ with the same $|A(\mathcal{R}_i)|$ as by following Rule 3.*

4.1.4 Establishing a Bound

Based on the rules for the C-RNLP, it is possible to bound the latest time at which a request could be satisfied.

Lemma 4.4. *Suppose that G is instantiated at time t and G' is instantiated at time t'. If $|A(\mathcal{R}_i)| > 0$, then $|A'(\mathcal{R}_i)| \leq |A(\mathcal{R}_i)| - (t' - t)$.*

Proof. The lemma follows because reservations are delay-preserving, and because all satisfied requests blocking $\mathcal{R}_i$ execute continuously between times t and t', due to their non-preemptive execution. □

Definition 4.18. Let $B_i = |A(\mathcal{R}_i)|$ at the time of $\mathcal{R}_i$'s insertion.

Lemma 4.5. *$\mathcal{R}_i$ is satisfied within B_i time units from its initial insertion.*

Proof. Follows directly from Lemma 4.4. □

Lemma 4.6. $B_i < N_i(L_{max} + L_i)$.

Proof. Suppose $\mathcal{R}_i$ requires only one resource. By Rule 3′, $\mathcal{R}_i$ cannot be inserted into a position P_k where $cap(P_k) < L_i$. In the worst-case scenario, $\mathcal{R}_i$ would have to be inserted into a top-most position with all other

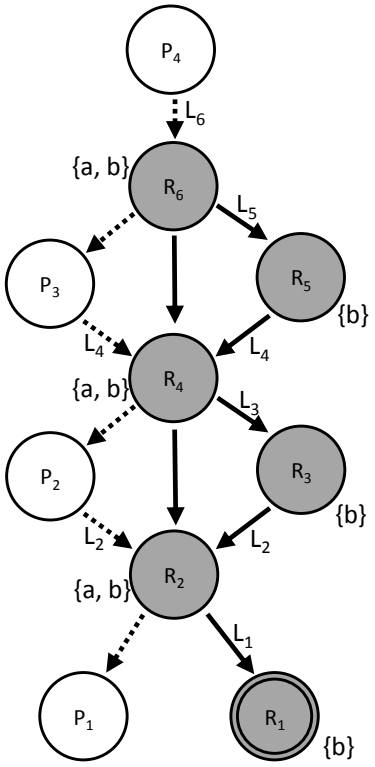


Figure 4.5: G with worst-case blocking for request $\mathcal{R}_i$ requiring ℓ_a, with L_i just greater than L_1, L_3, and L_5.

relevant positions having capacity just less than L_i. This scenario is shown in Figure 4.5, where $\mathcal{R}_i$ requires ℓ_a and L_1, L_3, and L_5 are all just less than L_i. In this case, $B_i < N_i L_{max} + N_i L_i$.

In the more general case, $\mathcal{R}_i$ requires a set of resources. Again, the worst-case scenario would only allow the insertion of $\mathcal{R}_i$ at a top-most position, with each other set of positions having capacity less than L_i. This scenario establishes the same bound. Therefore, $B_i < N_i(L_{max} + L_i)$. □

Theorem 4.1. *A request $\mathcal{R}_i$ is satisfied within $N_i(L_{max} + L_i)$ time units.*

Proof. Follows from Lemma 4.5 and Lemma 4.6. □

4.1.5 Uniform C-RNLP

Referring back to the proof of Lemma 4.6, when request $\mathcal{R}_i$ is inserted, a potential position for it can be *problematic* if that position has a non-zero capacity that is less than L_i. In the bound established in Theorem 4.1, the term $N_i L_i$ arises because there can be up to N_i problematic positions when $\mathcal{R}_i$ is inserted. This gives rise to a second variant of the C-RNLP, which is called the *uniform C-RNLP (U-C-RNLP)*, in which problematic positions cannot occur.

In this variant, the time line is segmented into fixed-length *frames* of size L_{max}. Wait-for graph modifications are allowed to occur only at frame boundaries, with all node removals occurring before any node insertions (still assuming that these graph modifications take zero time), and all satisfied requests are required to remain satisfied for exactly L_{max} time units. With these modifications, all parameters that affect the graph's basic structure are a multiple of L_{max}. Looking at Definition 4.15, notice that each position must have a capacity that is a multiple of L_{max}, or is 0, or is ∞. This implies that problematic positions cannot exist. As a result, the bound in Theorem 4.1 reduces to $N_i L_{max}$. However, because any request that occurs within a frame is delayed until the next frame boundary, this blocking bound must be increased by L_{max} (the frame size). This discussion results in the following theorem.

Theorem 4.2. *Under the U-C-RNLP, a request $\mathcal{R}_i$ is satisfied within $(N_i + 1)L_{max}$ time units.*

Corollary 4.1. *Under either variant of the C-RNLP, worst-case request blocking is $O(\min(m, c_i))$.*

Proof. Follows from Theorems 4.1 and 4.2, and the analysis assumptions and assumed progress mechanism of non-preemptive execution, which limits contention to at most m. □

Removal of assumptions. In the two variants of the C-RNLP just described, safety is maintained even if resource-holding times are longer than specified. In fact, the stated blocking bounds actually remain unaltered if a request $\mathcal{R}_i$ can hold a resource for longer than L_i time units, provided no request holds any resource for longer than L_{max} time units. However, if a request is allowed to hold a resource for longer than L_{max} time units, then the bounds no longer hold. From a practical point of view, this really is not a deficiency, because if reliable bounds on resource-holding times are not known, then it is pointless to attempt to conduct schedulability analysis.

The discussion so far has assumed that all modifications to a wait-for graph take zero time. In reality, of course, such modifications will entail some overhead. Such overheads can be factored into the blocking bounds using straightforward techniques. If these overheads are regarded as constants (as assumed in prior work on real-time locking protocols), then blocking times under both protocol variants remain contention sensitive.

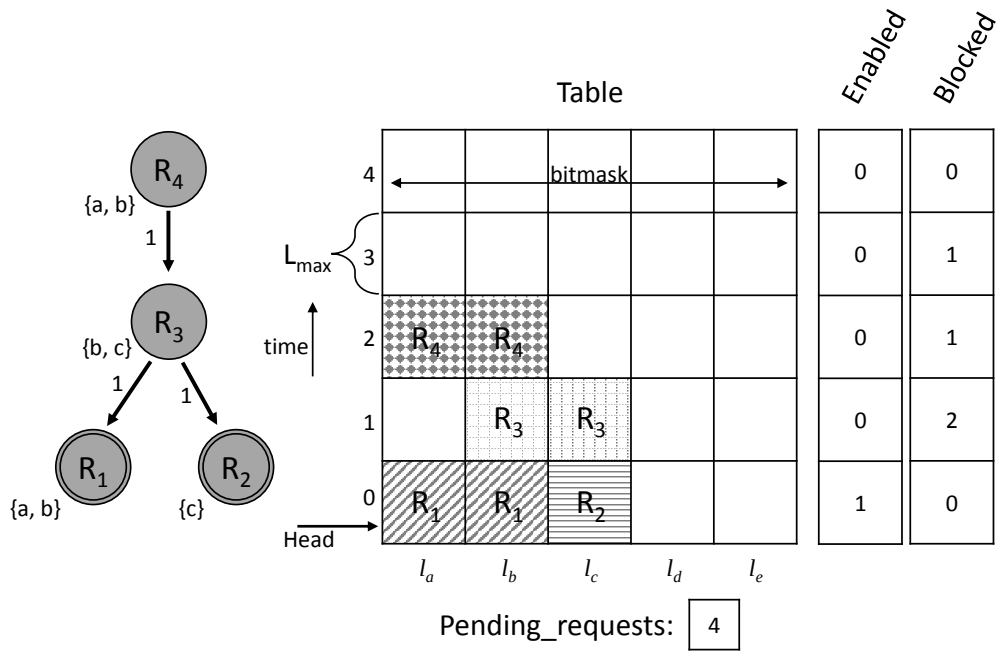


Figure 4.6: Illustration of *G* and *Table*.

4.2 Implementation

The prior section described two variants of the C-RNLP at an abstract level. This section presents a concrete implementation of the uniform variant.[2]

Data structures. Figure 4.6 depicts the key data structures of this implementation. A request acquires resources by reserving a set of positions in a reservation table, *Table*, which is an array of bit masks. Each bit in a bit mask represents a resource that can be reserved or locked for a *frame* of time of length L_{max}. Bit masks are used here because modern processors provide fast register-level operations for manipulating them. A request reserves a set of positions by selecting a row in *Table* (wrapping if necessary) and by setting the bits corresponding to its needed resources in that row's bit mask. The arrays *Enabled* and *Blocked* are both indexed by frame. A request $\mathcal{R}_i$ that has reserved a position given by row k is satisfied when $Enabled[k] = 1$ holds; such a row is said to be *enabled*. The manner in which *Blocked* is used is explained later. Several other variables are used as well. *Head* indicates the currently enabled row of *Table*. *Pending_requests* records the number of pending (issued but not completed) requests. *Size* gives the size of each array, which must be at least the number of pending requests at any time.

[2]Source code is available at `https://www.cs.unc.edu/~jarretc/dissertation/`.

Listing 6 Uniform C-RNLP Lock

```
 1: procedure U-C-RNLP LOCK(requested)
 2:     lock(Sublock)
 3:     if Pending_requests > 0 then
 4:         start ← (Head + 1) mod SIZE
 5:         while (Table[start] & requested) ≠ 0 do
 6:             start ← (start + 1) mod SIZE
 7:         end while
 8:     else
 9:         start ← Head
10:     end if
11:     next ← (start + 1) mod SIZE
12:     Blocked[next] ← Blocked[next] + 1
13:     Pending_requests ← Pending_requests + 1
14:     Table[start] ← Table[start] | requested
15:     unlock(Sublock)
16:     while Enabled[start] ≠ 1 do
17:         /* null */
18:     end while
19: end procedure
```

Pseudocode. The lock and unlock routines of this implementation are shown separately in Listing 6 and Listing 7, respectively. Note that shared variables are capitalized, while variables specific to a request or a function call are lowercase.

Figure 4.6 shows how a set of requests in a wait-for graph G would be represented in this implementation. As in G, requests that will be fulfilled later appear higher in *Table*. $\mathcal{R}_1$ and $\mathcal{R}_2$ are satisfied, as is indicated by $Enabled[0] = 1$. $\mathcal{R}_3$ has reserved ℓ_b and ℓ_c in $Table[1]$. The value 2 in $Blocked[1]$ indicates that $\mathcal{R}_3$ is waiting for both $\mathcal{R}_1$ and $\mathcal{R}_2$ to finish. $\mathcal{R}_4$ has similarly reserved ℓ_a and ℓ_b in $Table[2]$. The lock and unlock routines are now explained by means of examples.

Example 4.5. Suppose $\mathcal{R}_5$ for ℓ_b and ℓ_c is issued when the system state is as in Figure 4.6. First, a lock on *Sublock* is obtained at Line 2 of Listing 6 to serialize access to the protocol's shared state. For the request under consideration, the test in Line 3 would evaluate to true, so searching in *Table* for an available set of positions would begin. This search occurs in the loop at Lines 5 and 6, where the bit masks of different rows are checked. Upon termination of this search, the variable *start* indicates the corresponding row where the available positions were found. Following the search, *next* is set, $Blocked[next]$ and *Pending_requests* are incremented, and the appropriate row of *Table* is updated at Lines 11–14. The lock on *Sublock* is then released at Line 15. The task issuing the request then spins on $Enabled[start]$ at Lines 16 and 17. ◊

Listing 7 Uniform C-RNLP Unlock

```
20: procedure U-C-RNLP UNLOCK(requested)
21:     lock(Sublock)
22:     Table[Head] ← Table[Head] & ∼requested
23:     Blocked[next] ← Blocked[next] − 1
24:     Pending_requests ← Pending_requests − 1
25:     if Blocked[next] = 0 then
26:         Enabled[Head] ← 0
27:         Head ← next
28:         Enabled[Head] ← 1
29:     end if
30:     unlock(Sublock)
31: end procedure
```

Example 4.5 (continued). Now suppose that $\mathcal{R}_2$ completes. A lock on *Sublock* must first be obtained at Line 21 in Listing 7. At Line 22, *Table* is cleared of $\mathcal{R}_2$ by bit-wise ANDing the negation of $\mathcal{R}_2$'s *requested* bitmask and its row in *Table*. Note that *Head* now points to the acquired row of $\mathcal{R}_2$. *Blocked*[*next*] and *Pending_requests* are then decremented at Lines 23 and 24. If there are no more requests blocking requests in the row indicated by *next*, *i.e.*, *Blocked*[*next*] $= 0$ at Line 25, then that row is enabled and *Head* is updated at Lines 26–28. Finally, the lock on *Sublock* is released at Line 30. ♢

The above implementation limits the total number of resources to be at most the number of bits per bit mask, *e.g.*, on a 64-bit machine, there could be at most 64 resources in total. This restriction can be eased by applying results on the *renaming problem*, which is a classic problem in work on concurrent algorithms (Attiya et al., 1987). In this problem, tasks that have identifiers over a large name space are "renamed" by giving them identifiers over a small name space—such names can be both acquired and released (Moir and Anderson, 1995). A renaming algorithm could be applied in this context to assign a unique identifier to any resource while it is being used. This would require merely limiting the total number of *concurrently requested* resources to be at most the bit mask size. Renaming algorithms can be implemented with low overhead using appropriate atomic instructions. Alternatively, it is possible to extend this implementation by using several bit masks per row of *Table*, though this would increase lock/unlock overheads.

4.3 Experimental Evaluation

This C-RNLP implementation was evaluated by conducting a series of experiments in which lock/unlock overheads were measured, as well as observed blocking and runtime performance.

These experiments were performed on a dual-socket, 18-cores-per-socket Intel Xeon E5-2699 platform. The uniform variant of the C-RNLP was compared to the RNLP (Ward and Anderson, 2012) and Mellor-Crummey and Scott's queue lock (the MCS lock) (Mellor-Crummey and Scott, 1991a), applied as a coarse-grained lock, treating all resources as one resource group.

4.3.1 Measuring Lock/Unlock Overheads

Lock and unlock overheads (*i.e.*, the time it takes to perform the lock and unlock calls of each protocol) were measured as a function of the number of requested resources, *i.e.*, $|\mathcal{D}_i|$, the total number of managed resources, n_r, and the number of contending tasks, n. Tasks were statically pinned one per core.[3] The evaluation was comprised of the following variables: $n \in \{2, 4, \ldots, 36\}$, $n_r \in \{2, 4, 8, 16, 24 \ldots, 64\}$, and $|\mathcal{D}_i| \in \{1, 2, 4, 6, 8, 10\}$ where $|\mathcal{D}_i| < n_r$. Each contending task executed lock and unlock calls in a tight loop 1,000 times, with a negligible critical section, so as to maximize contention for shared variables within the lock and unlock calls.[4]

The C-RNLP lock and unlock calls themselves acquire a lock (which is true of the RNLP as well). Thus, blocking can occur within the lock/unlock logic. This blocking is part of the lock/unlock *overhead*, and we therefore refer to it as *overhead blocking*, to differentiate it from the *protocol blocking* (Lines 16 and 17 in the U-C-RNLP) experienced while waiting for C-RNLP-protected resources to be released. Therefore, when measuring lock/unlock overheads, we included overhead blocking in the measurement, but not protocol blocking. A similar methodology was applied to the RNLP and the MCS lock (the latter has no overhead blocking). These overheads were measured using the cycle counter, and the 99^{th} percentile of observed lock/unlock overheads is reported so as to filter any spurious measurements. Lock overhead is the focus of the results presented here; a similar story emerges when considering unlock overhead.

Figure 4.7 gives several curves pertaining to the lock-overhead data collected for the C-RNLP. Each curve is plotted with respect to the number of requested resources, $|\mathcal{D}_i|$. For the curve labeled *serial*, $\mathcal{D}_i$ was defined to be the same for all tasks (*i.e.*, resources are accessed serially). For the curve labeled *parallel*, $\mathcal{D}_i$ was determined at random (*i.e.*, resources can be accessed in parallel). These two curves include any overhead blocking. One might argue that it is better to account for such blocking analytically rather than by relying on measurement. To assess this possibility, we present two additional curves, *serial analytic* and *parallel*

[3]Tasks were pinned to cores on the same socket when possible.

[4]The source code and resulting graphs are available at `https://www.cs.unc.edu/~jarretc/dissertation/`.

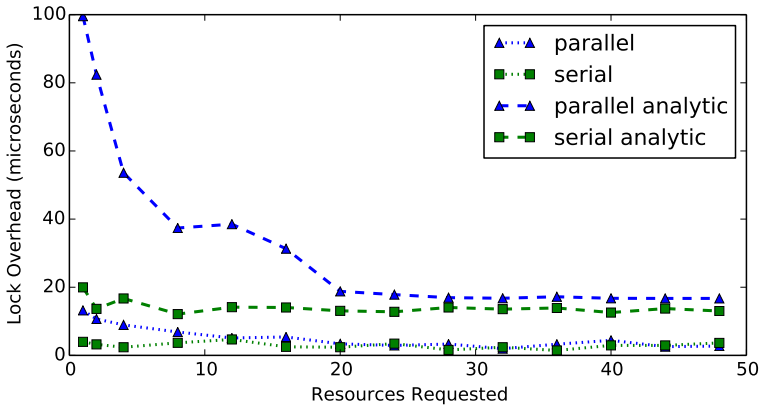


Figure 4.7: Measured C-RNLP lock overhead as a function of $|\mathcal{D}_i|$ for $n = 36$ and $n_r = 64$.

analytic, which were derived by measuring lock overhead with overhead blocking *excluded* and by inflating that measurement by accounting for overhead blocking analytically. Figure 4.7 leads to the following two observations.

Observation 4.1. The complexity of the lock logic in the C-RNLP requires an analytical estimation of worst-case overhead blocking as opposed to a purely measurement-based approach.

We observed a surprising overhead trend, supported by Figure 4.7, in the *parallel* case in that, as the number of requested resources $|\mathcal{D}_i|$ increased, the observed worst-case lock and unlock overheads *decreased*. In comparison to the *serial* case, the observed overheads were higher, which was also surprising given the potential that less of *Table* needed to be considered if more requests could be processed in parallel. Initially, we conjectured this was because the experimental process was not able to produce the worst-case overhead blocking. To address this, we considered overhead blocking analytically. The overheads using this analytical approach are indeed much higher, and therefore in a truly hard-real-time safety-critical system, an analytically rigorous approach must be taken to account for overhead blocking.

Interestingly, this observation has implications for other locking protocols that themselves employ a lock. In particular, suspension-based locks, which are implemented in the kernel, often acquire kernel-based spin locks. To truly bound the worst-case overheads of such protocols, a similar analytical approach should be taken to account for overhead blocking.

Observation 4.2. Runtime parallelism can increase worst-case lock and unlock overheads for the C-RNLP.

This can be seen by comparing the curves for the serial cases in Figure 4.7 to those for the parallel cases. We found the better performance in the serial cases quite surprising, because in these cases tasks do not

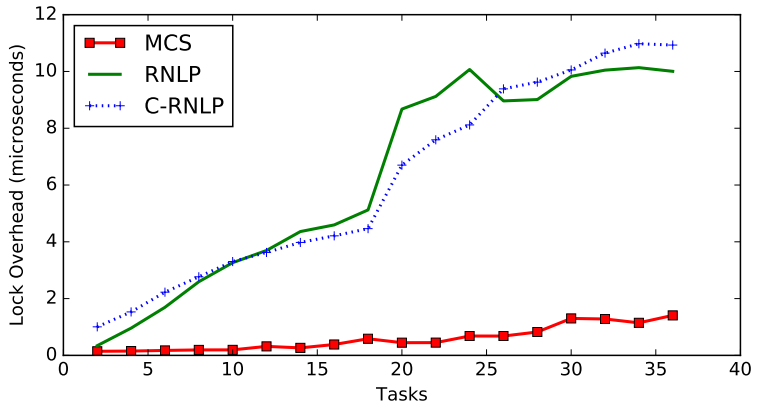


Figure 4.8: Lock overhead as a function of task count n for $n_r = 64$ and $|\mathcal{D}_i| = 4$.

"share" rows of *Table* and hence longer searches through *Table* are needed (indeed, we confirmed that less of *Table* was typically searched in the parallel case). However, at least in the context of this experimental framework, greater parallelism allows requests to be issued at a faster rate, since they experience less protocol blocking. This in turn increases contention for the shared variables in the C-RNLP implementation. We conjecture this increased contention resulted in cache invalidations and additional coherence traffic, such as inter-processor interrupts, that resulted in increased memory latency and therefore higher overheads.

The measurements discussed so far pertain only to the C-RNLP. Figure 4.8 plots measured lock overhead for all three considered protocols as a function of the task count n (recall that in the experimental framework, $n \leq m$). Two observations are supported by this data.

Observation 4.3. For the C-RNLP, observed overheads increased dramatically when resources were shared across sockets.

This observation applies to the RNLP as well. It can be confirmed by examining the sharp rise in the curves for both protocols between $n = 18$ and $n = 20$. This rise is due to increased memory latencies due to cross-socket interactions.

Observation 4.4. The fine-grained locking protocols had higher overhead than the coarse-grained one.

This can be easily seen in Figure 4.8. This result is not surprising, as coarse-grained protocols require far simpler lock/unlock logic. This exposes an interesting tradeoff: fine-grained protocols offer decreased blocking with higher overheads, while coarse-grained protocols offer decreased overheads at the expense of increased blocking.

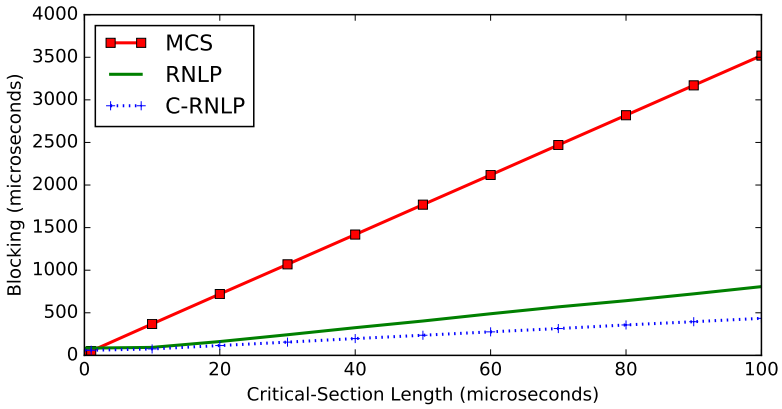


Figure 4.9: Total blocking time of lock call as a function of critical-section length for $n = 36$, $n_r = 64$, and $|\mathcal{D}_i| = 2$.

4.3.2 Runtime Performance

To examine this tradeoff, we measured the total lock and unlock overhead (including overhead blocking) and protocol blocking, which we hereafter simply refer to as *total blocking*. Specifically, we tested the same configuration parameters as in the previous experiments, and we also varied critical-section lengths within $\{1, 10, 20, \ldots, 100\}$ microseconds.[5] Figure 4.9 is a sample graph from this study, where $n = 36$, $n_r = 64$, and $|\mathcal{D}_i| = 2$. Based on these results, we make the following observation.

Observation 4.5. When critical-section lengths were greater than several microseconds and some parallelism was possible, the C-RNLP had less total blocking than previous protocols.

This can be seen quite dramatically in Figure 4.9, where both the RNLP and C-RNLP substantially outperform the MCS lock, with the C-RNLP (because of the greater parallelism it affords) besting the RNLP. Figure 4.9 was chosen to highlight the best-case scenario for the C-RNLP, *i.e.*, the case in which the most cutting ahead is possible. Obviously, for cases in which there is little if any cutting ahead (*e.g.*, the serial case described previously), the C-RNLP has inferior total blocking to the MCS lock as it results in the same request ordering, but with higher overheads. Additionally, in other scenarios there exist cases in which the overhead of the C-RNLP results in higher total blocking than the other protocols. These tradeoffs are explored more fully in the context of schedulability in Section 4.10.

[5]The resulting graphs are available at `https://www.cs.unc.edu/~jarretc/dissertation/`.

4.4 Motivation for the CGLP

The C-RNLP addressed the transitive blocking chain problem by reordering request satisfaction to allow newly issued requests to cut ahead in certain scenarios. However, the additional data that must be maintained by the locking protocol to ensure this is done safely results in overhead significantly higher than a simple lock like the MCS lock. This motivates reframing the problem of granting nested write access in order to realize a lower-overhead approach.

As described in Chapters 1 and 2, 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 the Transitive Blocking Chain Problem (Section 1.3). The second is a problem called the *Request Timing Problem*: even in protocols designed to reap gains in parallelism, such gains can be negated by 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.

The remainder of this chapter presents 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, it can instead be viewed it as a mechanism that *safely allows* concurrency with respect to shared resources.

The CGLP is designed around this new notion: groups of tasks that may safely execute concurrently. Before further description of the CGLP, the two fundamental problems are described in more detail.

4.4.1 Transitive Blocking Chain Problem

As described in Chapter 1, using a FIFO request ordering can result in chains of requests all blocked by a single request. Here, a small example illustrates how the Transitive Blocking Chain Problem can be solved by forming concurrency groups.

Example 4.6. 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 Figure 4.10, 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 Figure 4.10, $\mathcal{R}_1$ 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

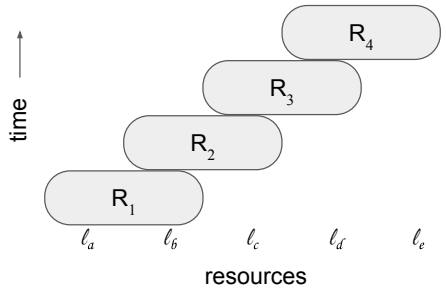


Figure 4.10: FIFO-ordering.

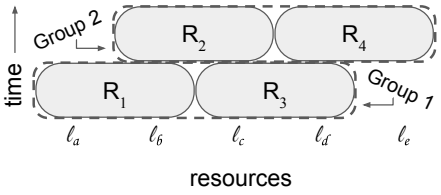


Figure 4.11: Optimized offline ordering.

Figure 4.10. Such a chain causes $\mathcal{R}_4$ to experience blocking for up to the duration of three critical-section lengths. ◊

The CGLP uses an alternative approach to determine the order of request satisfaction.

Example 4.6 (continued). To solve the Transitive Blocking Chain Problem, the CGLP partitions the requests in Figure 4.10 into two groups wherein concurrent execution is allowed, as shown in Figure 4.11. At runtime, resource access is provided on a per-group basis. As seen in Figure 4.11, doing so prevents transitive blocking chains from forming. ◊

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

4.4.2 Request Timing Problem

Although the C-RNLP has addressed the Transitive Blocking Chain Problem, worst-case blocking under it can be 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 4.7. Consider requests $\mathcal{R}_1$–$\mathcal{R}_4$, shown in Figure 4.12, issued in numerical order and enqueued. $\mathcal{R}_5$ is then issued and enqueued after $\mathcal{R}_4$. (Both C-RNLP variants result in $\mathcal{R}_5$ being satisfied after the completion of $\mathcal{R}_4$.) Another "slot" that could have been considered is shown in Figure 4.12, but $\mathcal{R}_5$ cannot be inserted here, as this would further delay $\mathcal{R}_4$. (Such delays are problematic because the number of later-arriving requests is generally unbounded.) Observe how the timing of the issuance of $\mathcal{R}_2$ caused a gap just after time 30 into which no conflicting request can fit. ◊

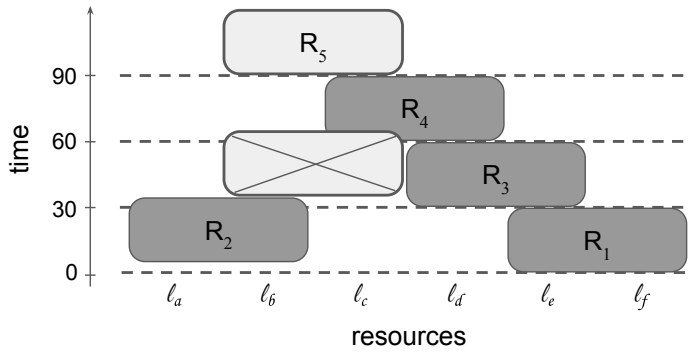


Figure 4.12: An illustration of the Request Timing Problem. $\mathcal{R}_5$ may not be inserted in the earlier slot marked by an 'X', as this would delay an already issued request.

In many protocols, having to deal with requests of different durations can also cause "gaps" similar to that in Example 4.7. Thus, such differences are also a source of the Request Timing Problem. The CGLP obviates such gaps by using task-system characteristics to pre-determine the "slots" into which requests are inserted. Because this determination is made offline, it is not subject to runtime timing variations and can account for duration differences.

This remainder of this chapter introduces the CGLP and describes its offline and online components. The offline portion (for which several options are presented) is analyzed on its execution time, and the online component on its overhead. Finally, the protocol as a whole is analyzed on the basis of schedulability.

4.5 Concurrency Groups

The Concurrency Group Locking Protocol (CGLP) is designed to address both the Transitive Blocking Chain Problem and the Request Timing Problem. Recall the pathological case of transitive blocking presented in Example 4.6. Although each nested request required only two resources, a FIFO-ordered synchronization protocol could cause a long chain of transitive blocking, as illustrated in Figure 4.10. The blocking chain in this example could be eliminated by partitioning the requests into the two groups shown in Figure 4.11 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.

This section begins by discussing how to generate concurrency groups for an arbitrary set of write requests. Next, a phase-based access protocol is extended to orchestrate all of the phases required by the concurrency groups. This section concludes by bounding the worst-case blocking any request may incur under the CGLP.

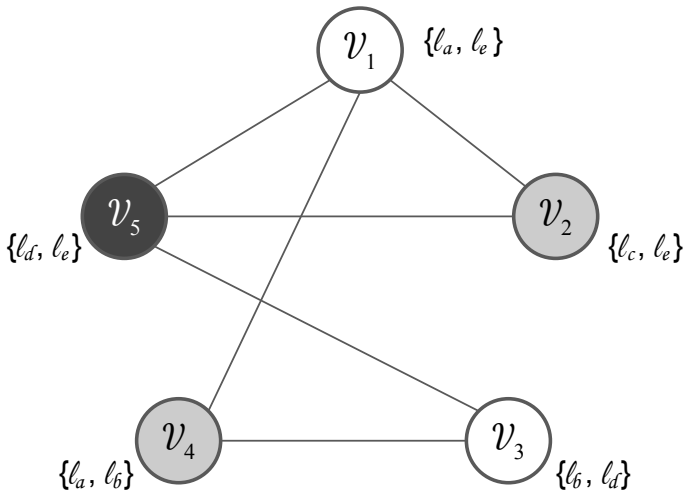


Figure 4.13: An example coloring.

4.5.1 Offline Group Creation via Graph Coloring

The concurrency groups are established by using a graph coloring approach. Such an approach has been used to solve a variety of other resource-allocation problems (Bandh et al., 2009; Barnier and Brisset, 2004; Chaitin et al., 1981; Marx, 2004).

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

When given a set of write requests, the offline component must be able to create concurrency groups. All requests in a single group must not share any resources. The goal is to create the minimum number of groups, as this maximizes the possible concurrency. This problem is transformed to the VCP in two steps. First, for each request $\mathcal{R}_i$, a corresponding vertex $\mathcal{V}_i$ is created. Once all vertices have been added to the graph, edges are added. 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 4.8. 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 Figure 4.13. 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 $\mathcal{D}_4 \cap \mathcal{D}_5 = \emptyset$. $\Diamond$

To determine the minimum number of concurrency groups, it suffices to 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 4.8 (continued). This graph is 3-colorable, so only three concurrency groups are required. In particular, the vertices can be colored as shown in Figure 4.13, 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 the 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 4.1. *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, it is assumed that all possible requests are known *a priori*. Thus, a k-colorability analysis can be run offline to determine the number of groups required for a given system and add each request to a group based on its assigned color.

4.5.2 Implementation of Offline Component

An instance of the VCP as an Integer Linear Program (ILP) can be encoded by using binary variables to indicate a color assignment for each vertex. The following formulation of this problem is based on a description in prior work (Palladino, 2010).

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

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

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

Using binary variables ensures $x_{i,c} \in \{0,1\}$. 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.

The 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.

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

Objective. $min \sum_c color_c$

To describe this problem as an ILP, a variable must be included for each available color. This requires pre-determining 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 Section 4.9).

Now let us specify the ILP corresponding to the running example from above.

Example 4.8 (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

$\Diamond$

Though the VCP is NP-hard, it is shown in Section 4.9 that, for many systems, groups can be determined in a reasonable amount of time. What remains is to coordinate access to these groups of requests during runtime.

4.5.3 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, synchronization must be provided between k groups of readers. This is achieved with a protocol called the R^{k}LP, which is a k-phased extension to the 3-phased reader-reader locking protocol (Chapter 3).

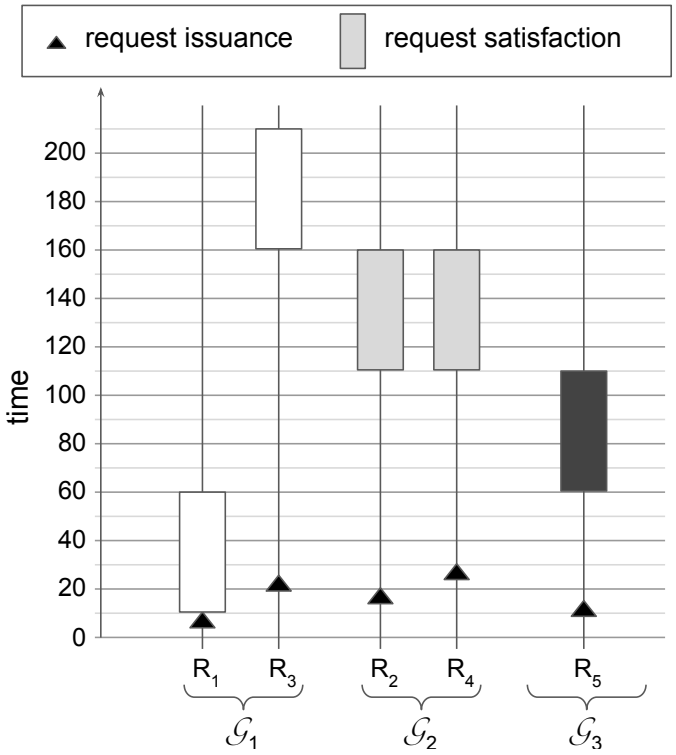


Figure 4.14: Trace of executions of requests in Example 4.8.

Example 4.8 (continued). No synchronization protection is required between requests $\mathcal{R}_1$ and $\mathcal{R}_3$, both in $\mathcal{G}_1$, as they do not share resources. However, $\mathcal{G}_1$ and $\mathcal{G}_2$ cannot be allowed to execute concurrently. ◊

The following rules encapsulate how the R^{k}LP functions. The time during which a group is active is called 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 4.8 (continued). As depicted in Figure 4.14, $\mathcal{R}_1$ 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 4.8 (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. ◊

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 4.8 (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. Next, the spin-based implementation of the R^{k}LP is discussed.

4.5.4 Implementation of Online Component

The R^{k}LP builds on the R^{3}LP (Chapter 3). Here the key components of the R^{k}LP implementation are described 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 if using a double-width compare-and-swap mechanism, 64 groups. While this may limit some applications, if the number of groups is larger than the number of processors, m, minimal analytical advantage can be gained by forming concurrency groups, as blocking under a non-preemptive, spin-based protocol is $O(m)$ (see Section 4.10.1). 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.

4.5.5 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, a bound on the worst-case acquisition delay is established.

Lemma 4.1. *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 4.2. *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 4.1. 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. □

Let us revisit the above example to see that this blocking bound is tight.

Example 4.8 (continued). When $\mathcal{R}_3$ is issued at $t = 25$ in Figure 4.14, 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 Figure 4.14. ◊

4.5.6 Refining the Blocking Bound

Up to this point, critical-section lengths were not specified, so each was treated as L_{max}. When requests have varying critical-section lengths, the bound in Theorem 4.2 may be overly pessimistic. When analyzing the impact of each concurrency group on the blocking a given request may experience, the maximum critical-section length of a group $\mathcal{G}_g$ is denoted $L_{max}^{\mathcal{G}_g}$.

Example 4.9. Here, let us use the same set of requests from Example 4.8, 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 4.2. *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.*

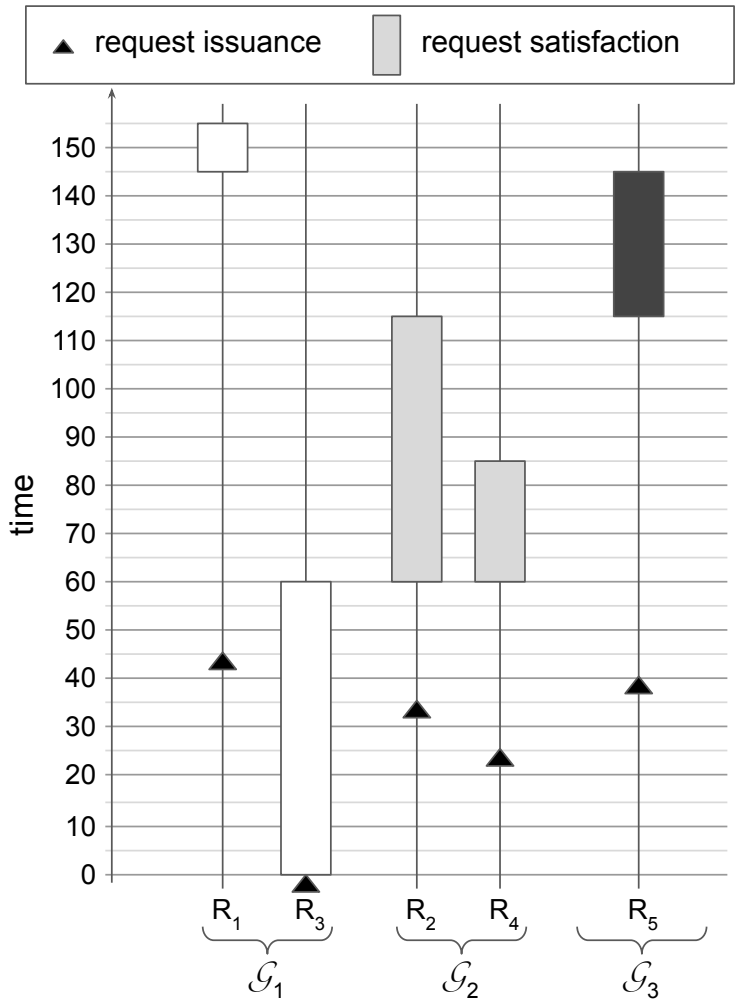


Figure 4.15: An illustration of the maximum blocking for $\mathcal{R}_1$ in Example 4.9.

Proof. As in Lemma 4.1, 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 4.3. *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 4.2, $\mathcal{R}_i$ 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 4.9 (continued). Consider the execution trace shown in Figure 4.15. 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 4.3, the worst-case

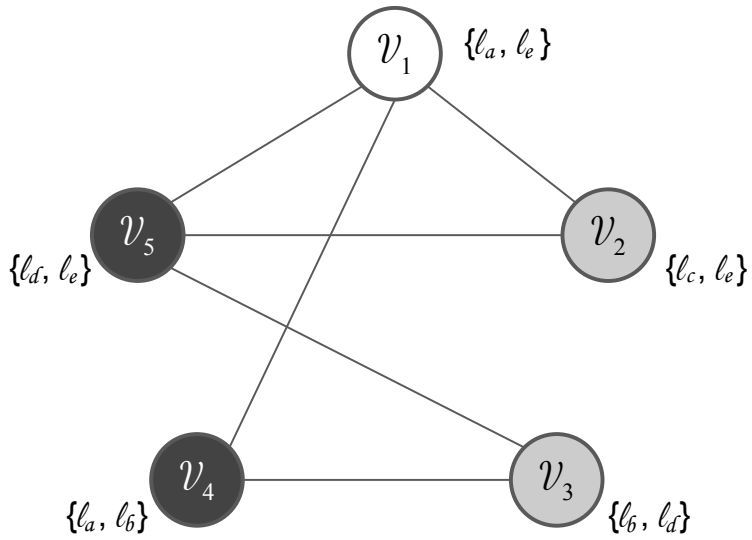


Figure 4.16: An alternate coloring.

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 4.2. ◊

4.6 Alternate Coloring Choices

Now that the fundamental components of the CGLP have been explained, several extensions to the protocol are discussed. In this section, the focus is 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, it is expected that this variant would outperform the basic CGLP from Section 4.5; the results of this comparison are presented in Section 4.10 (Observation 4.14).

4.6.1 Motivation

In the basic version of the CGLP, an arbitrary coloring of the vertices that required the minimum number of colors was chosen. However, there can be multiple ways to color a set of vertices with k colors, resulting in different concurrency groups. The following examples motivate a different method of grouping requests.

Example 4.9 (continued). Continuing the running example from the prior section, there are multiple ways to form concurrency groups for this set of requests. For example, instead of the coloring shown in Figure 4.13, the coloring shown in Figure 4.16 would yield $\mathcal{G}_1 = \{\mathcal{R}_1\}$, $\mathcal{G}_2 = \{\mathcal{R}_2, \mathcal{R}_3\}$, and $\mathcal{G}_3 = \{\mathcal{R}_4, \mathcal{R}_5\}$. ◊

$\mathcal{G}_1$	$\mathcal{G}_2$	$\mathcal{G}_3$	$\mathcal{G}_4$	$\sum_{c=1}^{4} L_{max}^{\mathcal{G}_c}$
$\mathcal{R}_1$	$\mathcal{R}_2$	$\mathcal{R}_3$	$\mathcal{R}_4$	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 4.1: Five possible groupings of the requests from Example 4.10 with the R^{k}LP blocking bounds computed. Using the minimum of two groups does not result in the lowest blocking.

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 4.3 when assigning groups.

Example 4.9 (continued). By Theorem 4.3, the worst-case blocking of any of the requests under the grouping shown in Figure 4.13 is $60+55+30=145$ time units. In contrast, the blocking under the grouping of Figure 4.16 is at most $10+60+30=100$ time units. Therefore, the grouping shown in Figure 4.16 should be used instead of that in Figure 4.13. ◊

Example 4.9 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 the primary goal may require more than k groups. This is illustrated with an example.

Example 4.10. 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. Table 4.1 shows all possible groupings of these requests; those with different group number assignments are simply permutations of these groups and result in the same summed blocking. ◊

Example 4.10 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 Figure 4.17.

4.6.2 Minimizing Blocking

To minimize the impacts of synchronization on the system, it is possible to instead minimize blocking with the assignment of requests to concurrency groups. This is achieved by developing an ILP that follows

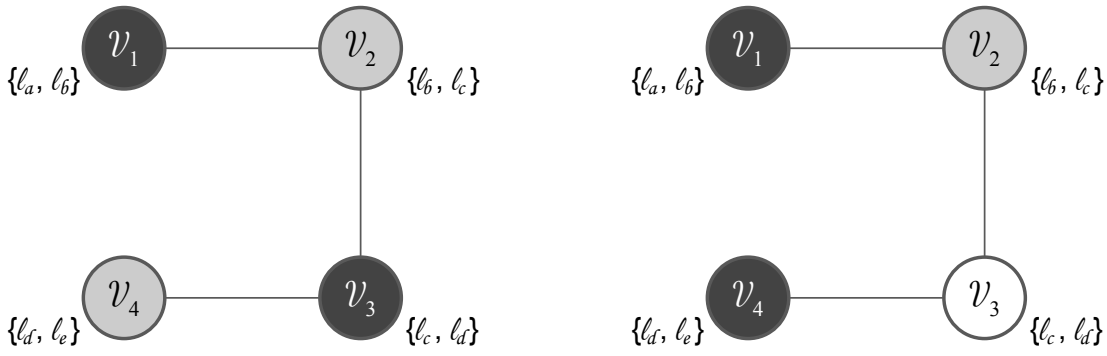


Figure 4.17: For the requests in Example 4.10, the corresponding minimum coloring is on the left, and the coloring that achieves the minimum blocking is on the right.

many of the same principles as the ILP presented in Section 4.5.2. First, the variables are described, and then the details of the constraints are given.

Variables. As in Section 4.5.2, the notion of graph coloring to guides the approach. The binary variable $x_{i,g}$ is used to indicate whether $\mathcal{V}_i$ is assigned Color g (*i.e.*, $\mathcal{R}_i$ belongs to $\mathcal{G}_g$). To capture $L_{max}^{\mathcal{G}_g}$, the variable $duration_g$ is used. Here, the optimization problem is no longer to 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. Constraints 1 and 2 from Section 4.5.2 are used here and are restated below.

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

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

The following constraint 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.

The objective function is to minimize $\sum duration_c$ over all possible colors. This minimizes overall blocking, as computed by the expression in Theorem 4.3.

Objective. $min \sum_c duration_c$

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

Example 4.10 (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

$\Diamond$

4.7 Mixed-Type Requests

Recall from Section 2.4.1 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. These different synchronization requirements can be captured in a manner that enables exploiting the relaxed resource-sharing assumptions for read requests. This is achieved by modifying how graphs are generated relative to the requests.

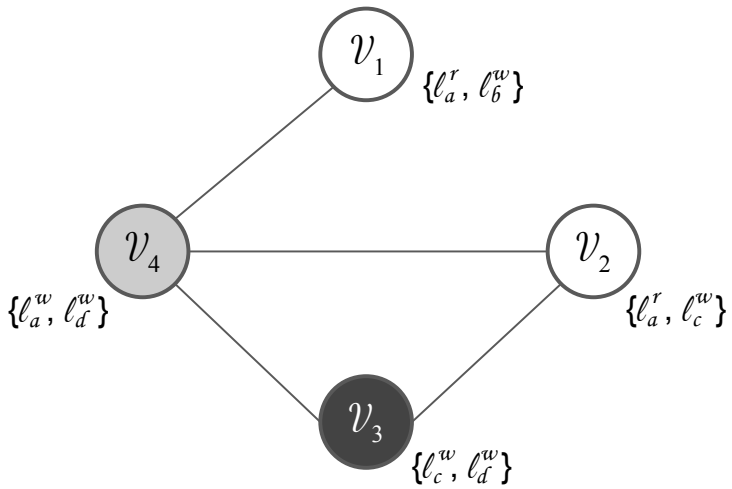


Figure 4.18: Graph of mixed-type requests.

4.7.1 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 $\mathcal{R}_i$, the type of access required (read or write) is denoted 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 4.11. 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. ◊

The set $\mathcal{D}_i^w = \{\ell_y | \ell_y^w \in \mathcal{D}_i\}$ is 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 4.11 (continued). The graph corresponding to this set of requests is shown in Figure 4.18. 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 Section 4.5.6 can be applied.

4.7.2 Modifications to ILP

Both of the ILPs 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 Section 4.6.2, the number of colors in the ILP is given by the number of vertices in the graph.

Example 4.11 (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

$\Diamond$

4.8 Hierarchical Organization

The initial approaches to determining concurrency groups resulted in each request being satisfied with the same frequency. This section explores adding a layer of hierarchy to the request-management scheme to alter the frequency with which requests are satisfied. Let us begin by considering a group of six requests: the five requests from Example 4.9 and one additional request.

Example 4.12. Consider the task set with the requests from Example 4.9 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 Sections 4.5.1 and 4.6.2, we determine that four concurrency groups, as shown in Figure 4.19, are required (adding $\mathcal{G}_4 = \{\mathcal{R}_6\}$ to the three groups used in Example 4.9). This grouping results in worst-case blocking for all requests of $10 + 60 + 30 + 55 = 155$ time units.

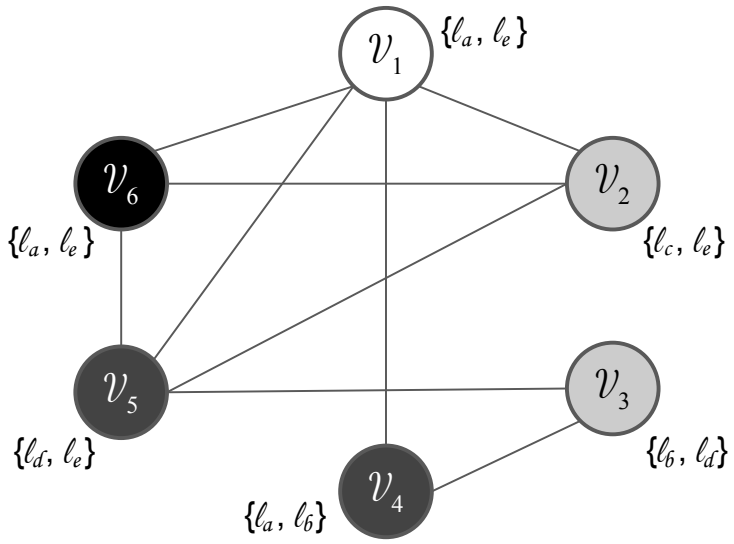


Figure 4.19: Four concurrency groups for requests $\mathcal{R}_1$ to $\mathcal{R}_6$: $\mathcal{G}_1 = \{\mathcal{R}_1\}$, $\mathcal{G}_2 = \{\mathcal{R}_2, \mathcal{R}_3\}$, $\mathcal{G}_3 = \{\mathcal{R}_4, \mathcal{R}_5\}$ and $\mathcal{G}_4 = \{\mathcal{R}_6\}$.

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 Example 4.9, each request in this example may experience 155 time units of blocking. ◊

This section proposes 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.

4.8.1 Hierarchical Request Satisfaction

Under this scheme, a set of *slots* becomes active in a round-robin-like fashion like the group arbitration process described in Section 4.5.3. 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 Section 4.5.3, but now a group competes with only the other groups in the same slot. Each group belongs to exactly one slot. The set of all groups belonging to Slot a is denoted $\mathcal{S}_a$.

Example 4.12 (continued). The concurrency groups depicted in Figure 4.19 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. The rules below add to those stated in Section 4.5.3 in order to capture this new structure. All of the rules in Section 4.5.3 apply without modification

(*e.g.*, note that G8 refers to any waiting group belonging to any slot) except for G3, which is replaced with a modified version below.

The first set of rules are those 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.

These rules are illustrated with an example depicted in Figure 4.20 and described below.

Example 4.12 (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. $\Diamond$

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 4.12 (continued). As shown in Figure 4.20, 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 $t = 95$. $\Diamond$

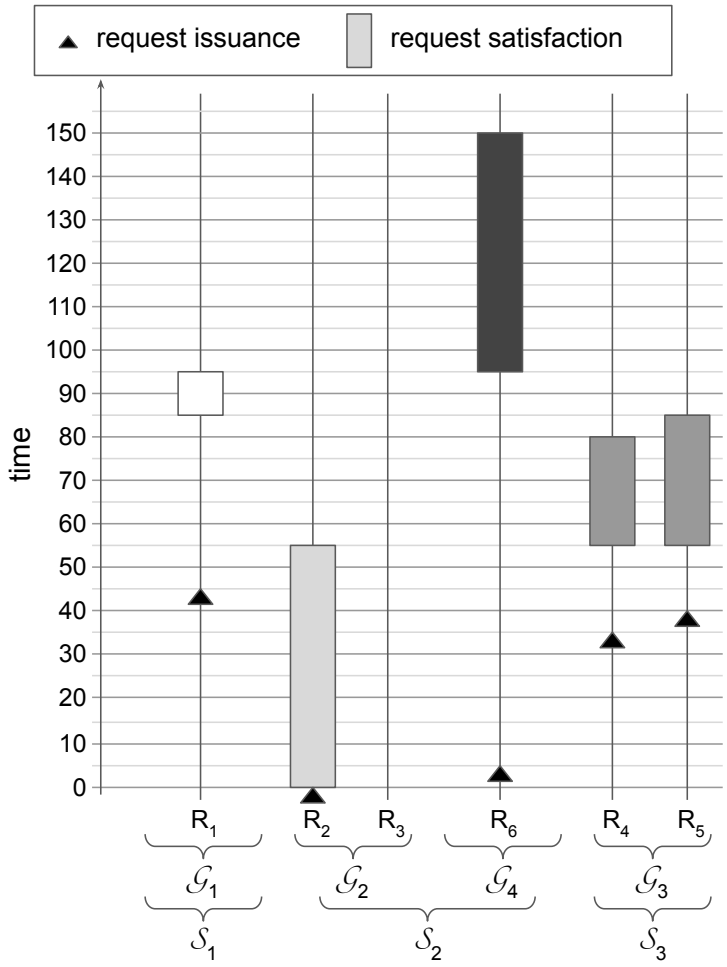


Figure 4.20: An illustration of execution under the hierarchical approach.

As described in the rules above, a FIFO ordering among groups competing for a given slot is enforced; 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.

4.8.2 Bounding Blocking

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

Lemma 4.3. *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 4.12 (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 4.3, this will occur within $60 + 30 = 90$ time units. ◊

Theorem 4.4. *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 4.3 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 4.3 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 4.3 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 4.3 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 4.3 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 4.12 (continued). As depicted in Figure 4.20, 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 4.4: $\mathcal{R}_6$ 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, it is possible to 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.

4.8.3 Assigning Groups to Slots

Although Theorem 4.4 upper-bounds the acquisition delay each request may experience, it does not guide how to assign groups to slots. The following provides some intuition behind assignment decisions and a few possible approaches. In general, the benefit of adding a layer of hierarchy in Example 4.12 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 (*e.g.*, with the ILP in Section 4.7.2). 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 presented here 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. This is specified as an optimization problem, and the approach is described in more detail as each constraint is presented.

As before, a maximum number of colors must be encoded. This is chosen to be the number of request-issuing tasks, denoted *num_req*. Here, groups are pre-selected to 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 $\mathcal{S}_{\lfloor \frac{g}{2} \rfloor + (g \bmod 2)}$ if $g \leq split$ or $\mathcal{S}_{g - \frac{split}{2}}$ otherwise. Alternative choices of *split* may also be considered.

Now the constraints that are described. The constraints from Section 4.5.2 to enforce the basic coloring restrictions are reused here.

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 a set of variables is used to represent the maximum duration of each slot: $duration_a$ represents $L_{max}^{\mathcal{S}_a}$. Based on the construction of this 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. $\forall i \; \forall a : a > \frac{split}{2} : duration_a \geq x_{i,a+\frac{split}{2}} \cdot L_i$

The next constraint limits the variable *summed_duration*, which upper-bounds the sum in Theorem 4.4.

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

The 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 4.4.

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

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

The objective of this optimization problem is 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 Section 4.7 to handle mixed requests.

4.9 Analysis of Offline Component

The 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. This section focuses on evaluating the offline portion.

Here, three possibles approaches to implementing the offline component of the CGLP are evaluated. Section 4.5.2 presented an ILP to assign concurrency groups such that the number of groups is minimized. This approach is denoted BasicILP. Section 4.6.2 presented an optimization problem to instead minimize blocking, denoted here as DurationILP. Finally, Section 4.7.2 showed how the above two optimization problems can be modified to account for mixed requests in a more fine-grained manner. Here, this modification is applied to the duration-based ILP, and the resulting approach is denoted RW-DurationILP.

These offline components are compared on the basis of how long determining the concurrency groups takes across a range of task systems. In this evaluation, task systems were generated across broad space of task-system parameters, varying the individual task utilization, the period, the percentage of tasks that issue

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

Category	Options
Task Utilization	Medium, Heavy
Period	Short, Long
Percentage Issuing Requests	50%, 80%, 100%
Critical-Section Length	Moderate, Bimodal, Weighted Bimodal, Long
Number of Resources	64
Nested Probability	0.1, 0.2, 0.5
Mixed Probability	0, 0.2, 0.5, 0.8
Nesting Depth	2, 4

Table 4.3: 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.

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 4.2, and the set of parameters used for the evaluation are in Table 4.3. 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. A *scenario* is defined to be a setting of each of the above parameters. These parameter selections correspond to those used for the schedulability study in Section 4.10.

To give a sense of the scale of these problems, Table 4.4 reports the average number of requests and average minimum number of colors (using BasicILP) across 50 task sets. These task sets had a total utilization of 16, every task issued a request, nested probability was varied, and nested requests required four random resources. Note that for task sets with heavy per-task utilization, fewer tasks are needed to reach the given

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 4.4: Average size of a graph coloring problem for a system with total utilization of 16.

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 the specification of each ILP, variables for each color that might be needed must be created. In order to improve performance of BasicILP, this number can be restricted (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 tighter upper bound on the chromatic number than simply using the total number of requests. Here, the greedy approach 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, random task sets across the space of all scenarios were generated. For each task set, the setup time and the time required to solve each ILP with an ILP solver (Gurobi, 2018) was measured. Figure 4.21 shows 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 mixed was 0.8. For each scenario, 100 task sets were generated.

Observation 4.6. Although the connection of the problem of determining groups to the NP-hard 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 Figure 4.21, 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, a timeout on the ILP solver was set for each ILP when running the schedulability study (described in more detail in Section 4.10). For task sets with heavy per-task utilization, 0.1s was used for

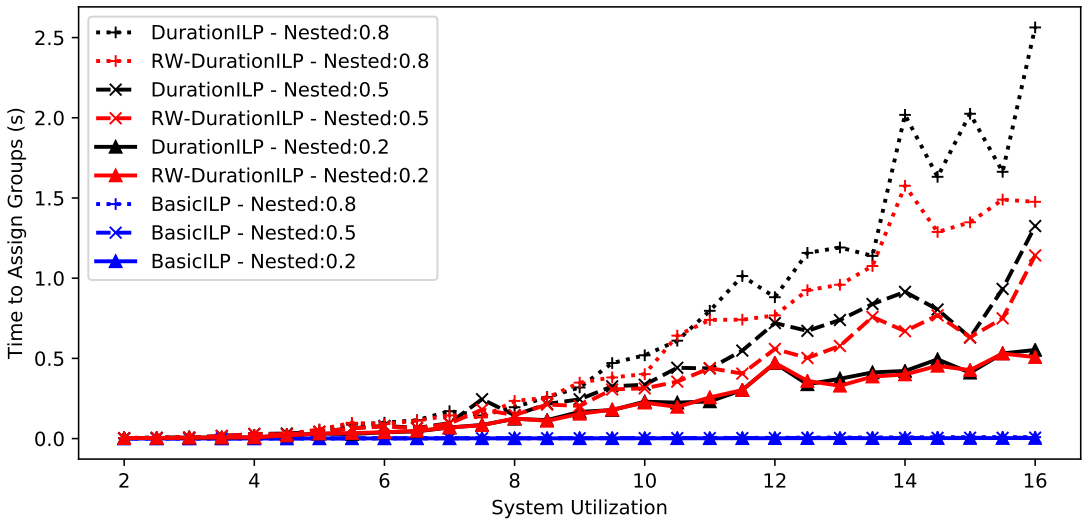


Figure 4.21: Average time to solve each ILP. Each data point represents the 95$^{\text{th}}$ percentile from 100 random task sets.

BasicILP and 2.7s for the two duration-based ILPs. In the schedulability experiments, these thresholds were never exceeded. For task sets with medium per-task utilization, a timeout of 0.1s was enforced for BasicILP and 120s for the duration-based ILPs. Across 16.7 million task sets, one of these limits was exceeded only 153 times.

Observation 4.7. The time to solve each ILP depended 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. This is illustrated this with a small set of scenarios: Figure 4.21 shows the time required to solve each ILP for a scenario in which each nested request requires four resources. As shown in Figure 4.21, 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 were significantly higher than those for BasicILP. We hypothesize that the solution times for DurationILP and RW-DurationILP were higher partially due to having so many additional variables; the greedy coloring could not be used to determine the minimum number of colors required to enable the ILP to minimize blocking. (Recall the discussion in

Section 4.6.1 that minimizing colors may not minimize duration.) Therefore, the number of requests was used as a safe upper bound on the number of colors for the duration-based ILPs.

4.10 Schedulability Study

This section presents an evaluation of the C-RNLP and the CGLP on the basis of schedulability with a large-scale schedulability study. These protocols were compared to existing protocols across a variety of task sets. The C-RNLP and the CGLP were compared to the RNLP and to simple group lock, for which a single MCS lock (Mellor-Crummey and Scott, 1991a) is used to protect all resources. For the C-RNLP, both the uniform variant (the U-C-RNLP) and the general variant specified by the rules of the protocol (the G-C-RNLP) are considered.

4.10.1 Experimental Setup

The following schedulability experiments were conducted with SchedCAT (SchedCAT, 2019), an open-source real-time schedulability test toolkit. SchedCAT was used to randomly generate task systems, compute blocking bounds, and determine schedulability on a 16-core platform under G-EDF scheduling as described in Chapter 2. The execution time of each task was inflated based on the locking protocol overhead and blocking its requests may incur, as described in prior work (Brandenburg, 2011). The range of task systems considered in this evaluation is given in Table 4.3.

The blocking bounds and overhead of each protocol are summarized in Table 4.5. 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; the results for three of them are shown here. A timeout value was set for the ILP solver (see Section 4.9) for each approach. If this limit is ever exceeded, groups are instead assigned with a greedy coloring approach.

When calculating the blocking bounds for the CGLP in the schedulability study, the expression presented in Theorem 4.3 was used, along with additional refinements to bound the acquisition delay $\mathcal{R}_i$ can experience.

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

Table 4.5: Blocking bounds and overhead of each protocol. For the C-RNLP bounds, N_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.)

For example, because the focus here is 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 are 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 is used.

Similar refinements can be applied to the C-RNLP variants. Again, at most $m-1$ other requests may be active for each variant. Tighter analysis is also applied to determine the longest critical sections that may block $\mathcal{R}_i$; this is different between the two variants, but for both it is possible to 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 $\mathcal{R}_i$.

The schedulability study considered the 1,152 scenarios listed in Table 4.3. For each scenario, 1,000 task systems were generated for every value of system utilization; the percentage of these that are schedulable is plotted as the HRT Schedulability. Each plot includes a curve for when no synchronization delay is accounted for (NOLOCK) and when synchronization delay results from each of the protocols compared.[6]

To compare the schedulability results under different protocols, the *schedulable utilization area* (SUA) is computed for each by approximating the area under the curve with a midpoint Riemann sum. When comparing protocols, 0.05 was used as a threshold for determining if two protocols performed about as well

[6]The full set of plots and the source code are available at `https://www.cs.unc.edu/~jarretc/dissertation/`.

as each other; if their SUAs in a given scenario differed by less than 0.05, they are reported to have performed roughly the same under that scenario.

The following sections present general observations from the schedulability study along with graphs that showcase key trends.

4.10.2 Evaluation of the C-RNLP Variants

This evaluation begins by considering the 288 scenarios in which the probability of a nested request being mixed is zero; the C-RNLP and the existing protocols do not handle mixed requests. The results from these scenarios result in the following observations, which are illustrated in Figure 4.22 with four scenarios. Each point on a protocol curve depicts the fraction of task sets at that system utilization that were deemed schedulable when the analysis for the given protocol was applied.

Observation 4.8. The C-RNLP variants outperformed the group MCS lock.

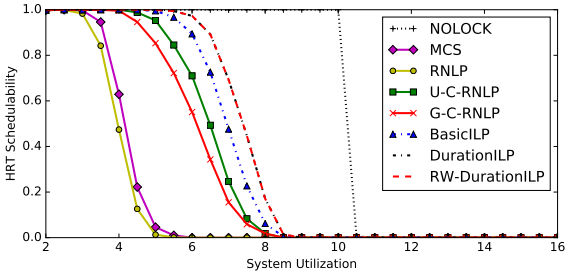
The U-C-RNLP resulted in better SUA than MCS in 90.3% of scenarios, and the G-C-RNLP resulted in better SUA than MCS in 91.0% of scenarios. Figure 4.22 illustrates a few of the scenarios in which the C-RNLP variants outperformed the group lock.

Observation 4.9. The C-RNLP variants outperformed the RNLP.

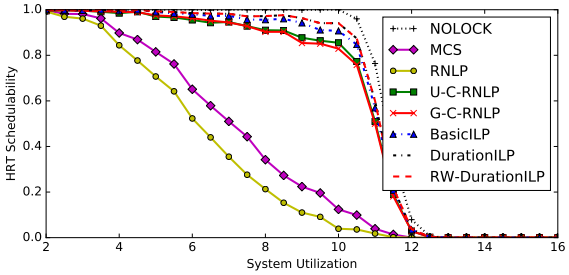
The U-C-RNLP resulted in better SUA than MCS in 91.0% of scenarios, and the G-C-RNLP resulted in better SUA than MCS in 91.7% of scenarios. Again, this is illustrated by the scenarios in Figure 4.22.

Observation 4.10. The G-C-RNLP tended to perform as well or better than the U-C-RNLP.

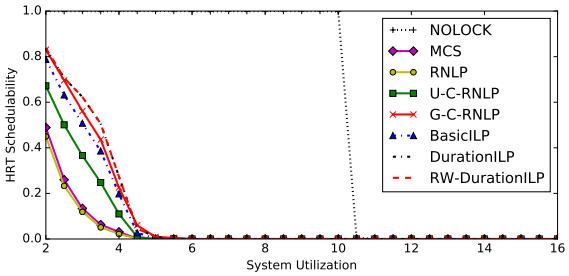
The G-C-RNLP resulted in a higher SUA than the U-C-RNLP in 49.3% of scenarios, and the U-C-RNLP resulted in a higher SUA than the G-C-RNLP in only 17.0% of scenarios. The scenarios in which the U-C-RNLP resulted in a higher SUA tended to be those that required more resource access (higher request probability, nested probability, and number of resources requested), and many of these were scenarios with moderate critical-section lengths. One such scenario is shown in Figure 4.22 (a). In fact, the G-C-RNLP resulted in a higher SUA than the U-C-RNLP in only 15.3% of the scenarios with moderate critical-section lengths.



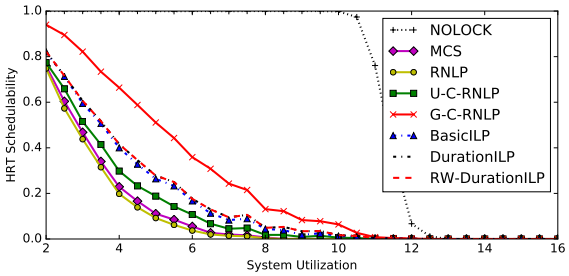
(a) Medium utilization, moderate critical-section length



(b) Heavy utilization, moderate critical-section length



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



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

Figure 4.22: 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.

4.10.3 Comparison of the CGLP to Existing Protocols

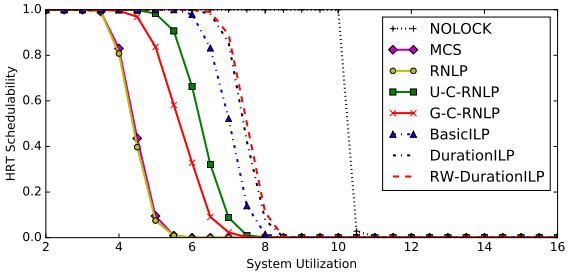
The CGLP was also compared to the C-RNLP variants and existing protocols for the 288 scenarios without mixed requests. These comparisons resulted in the following observations.

Observation 4.11. The CGLP approaches performed as well as or better than the RNLP or the MCS in all scenarios.

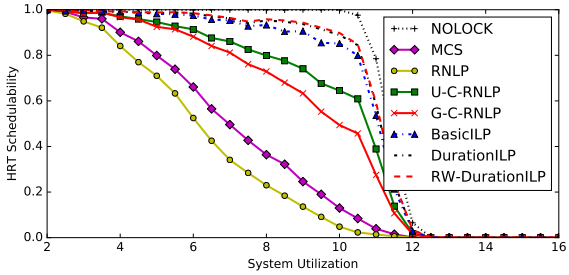
This is illustrated in Figure 4.22 (a-d). The CGLP resulted in a higher SUA than both the RNLP and the MCS in 91.3% of scenarios.

Observation 4.12. The CGLP approaches performed as well as or better than the C-RNLP variants in many scenarios.

This is illustrated in Figure 4.22 (a-c). The CGLP resulted in the highest SUA (tied or alone) in 66.3% of scenarios. For task systems with moderate critical-section lengths, the CGLP approaches always had the highest SUA (tied or alone). For medium per-task utilization, the CGLP approaches performed as well or



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



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

Figure 4.23: For this scenario, nested probability was 0.5, nesting depth was 4, mixed probability was 0.8, and 100% of tasks issued requests.

better than other approaches in 70.1% of scenarios, and in 54.9% of scenarios, the CGLP outperformed both C-RNLP variants.

Observation 4.13. When a CGLP approach was not the best, the G-C-RNLP was the protocol that resulted in the highest SUA.

In 33.7% of scenarios, the G-C-RNLP outperformed all other protocols. In 97.9% of these scenarios, periods were short. One such scenario is illustrated in Figure 4.22 (d).

4.10.4 Comparison of CGLP Variants

In this section, the different offline CGLP components are compared. This evaluations considers all values of mixed probability used in the schedulability study. Note that the analysis of existing protocols and that of the C-RNLP variants does not handle mixed requests and instead must treat such requests as write requests. The following observations are based on this study, and the results of two scenarios are highlighted in Figure 4.23. Both of these scenarios had a nested probability of 0.5, a nesting depth of 4, a mixed probability of 0.8, and 100% of the tasks issued requests.

Observation 4.14. DurationILP was never worse than BasicILP; frequently it was better.

DurationILP was better in 52.5% of scenarios. In each scenario in Figures 4.22 and 4.23, DurationILP is better than BasicILP, as determined by comparing SUAs.

Observation 4.15. While RW-DurationILP frequently had a slightly higher SUA than DurationILP, it rarely was 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 Figure 4.23 (a). A scenario in which the SUA of RW-DurationILP exceeds that of DurationILP by less than 0.05 is shown in Figure 4.23 (b).

4.11 Chapter Summary

This chapter presented two variants of the C-RNLP, the first multiprocessor real-time locking protocol to be contention sensitive in the presence of nested requests. The C-RNLP incorporates knowledge of critical-section lengths to order requests in a way that breaks long transitive-blocking chains. This technique increases lock and unlock overheads. However, in the context of the schedulability study presented herein, this tradeoff of higher overheads still tended to result in better performance for the C-RNLP variants than existing protocols.

Additionally, this chapter presented the CGLP, a real-time locking protocol that grants nested resource access and solves both the Transitive Blocking Chain Problem and the Request Timing Problem. The offline component of the CGLP determines concurrency groups and simplifies the arbitration of requests at runtime, reducing overhead. Several versions of the offline component were presented, each of which examines optimizations to the request ordering and yields analytical blocking advantages. These options were evaluated with a schedulability study that found that the CGLP outperformed other protocols in most scenarios.

Acknowledgements. The work presented in this chapter originally appeared in several papers. The C-RNLP was developed in a paper by Bryan Ward and myself. Bryan guided the development and implementation of the protocol and helped plan the experiments. The CGLP was presented in papers written by Tanya Amert, Manish Goyal, and myself. Tanya was involved in the development of the CGLP, implemented the G-C-RNLP, ran the overhead experiments, and assisted with the schedulability study. Manish implemented the linear programming framework for the offline component of the CGLP.

CHAPTER 5: LOCK SERVERS[1]

The two C-RNLP variants provide contention-sensitive blocking by effectively "breaking" transitive blocking chains. Unfortunately, the complex protocol logic required to enable such blocking can result in high overhead.

To mitigate this issue, this chapter considers the usage of lock servers to reduce overhead. A *lock server* is a special process that sequentially performs all lock and unlock functions of a given protocol. The main advantage of using lock servers is that they can run cache hot (which is explained in the context of an example platform in Section 5.1). The main disadvantage is the need to dedicate whole cores, or fractions of cores, to performing synchronization functions. However, on machines with high core counts, this may be a reasonable thing to do, as has been observed by others in other contexts (Hsiu et al., 2011; Lozi et al., 2012).

A number of server-based locking protocols employ notions similar to a lock server but for the purpose of easing the calculation of pi-blocking bounds. The first such protocol was the DPCP (Rajkumar, 1990, 1991; Rajkumar et al., 1988), which statically binds resources to cores and requires tasks to perform lock and unlock calls for a resource on the core assigned to that resource. Subsequently, a number of server-based protocols were proposed that follow a similar approach (Burns and Wellings, 2013; Faggioli et al., 2010, 2012; Hsiu et al., 2011; Huang et al., 2016; Lakshmanan et al., 2009; Zhao et al., 2017). In contrast to these server-based protocols, the goal of lock servers is to preserve the blocking bounds of a given protocol while reducing its overhead.

Lock servers were partially inspired by work on a concept called *remote core locking (RCL)*, which was directed at improving the performance of legacy *non-real-time* code when moving it from a uniprocessor system to a multiprocessor one (Lozi et al., 2012). In particular, RCL seeks to avoid cache-line bouncing when a resource is accessed on different cores by requiring all resource accesses to occur on a designated

[1]Contents of this chapter previously appeared in preliminary form in the following paper:

Nemitz, C., Amert, T., and Anderson, J. (2018). Using lock servers to scale real-time locking protocols: Chasing ever-increasing core counts. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems*.

core. The emphasis in work on RCL is to enable critical sections to run cache hot. In contrast, the goal with lock servers is for lock and unlock routines to run cache hot.

This chapter presents four lock-server paradigms that are defined by specifying servers as either static or floating and either global or local. A *static* lock server is bound to a single core (Section 5.1), while a *floating* one may migrate (Section 5.2). Lock servers are also specified by their locality: a *global* lock server handles requests from all cores, while a *local* one handles requests from only its socket. These paradigms are initially presented as they are applied to the U-C-RNLP, with a subsequent discussion of the changes required for application to the G-C-RNLP (Section 5.3). Next, an experimental evaluation of each of the proposed approaches is given (Section 5.4). Finally, additional details on the coordination between local lock servers is given (Section 5.5).

5.1 Static Lock Servers

This sections considers the use of static lock servers to implement the U-C-RNLP. The distinctions of the lock server paradigms are given in the context of a 36-core test platform.

Platform description. In order to describe the lock-server paradigms considered in this chapter more concretely, their application on a particular test platform, a dual-socket, 18-cores-per-socket Intel Xeon E5-2699, is the focus. This platform provides significant per-socket parallelism while allowing issues on a multi-socket machine to be explored. As depicted in Figure 5.1, each core has a 32KB L1 data cache and a 32KB L1 instruction cache. Pairs of cores share a unified 256KB L2 cache, and all cores on a socket share a unified 45MB L3 cache. Lock state is considered to be *cache hot* if it maintains cache affinity in the lowest-level cache shared among all cores on which the server may execute. Keeping the lock state cache hot can reduce access time and cache coherence traffic, such as inter-processor interrupts.

The problem. Before delving into some of the nuances of using lock servers, let us examine the problem that they are intended to solve. Figure 5.2 plots lock overhead as a function of core count (and thus number of requests) for three possibilities: the U-C-RNLP as originally presented; the same protocol but implemented using a single global lock server (denoted U-C-RNLP + SGLS); and an implementation in which all resources are coalesced under one lock using Mellor-Crummey and Scott's queue lock (denoted MCS) (Mellor-Crummey and Scott, 1991a). The MCS is taken as the gold standard for low overhead. Notice

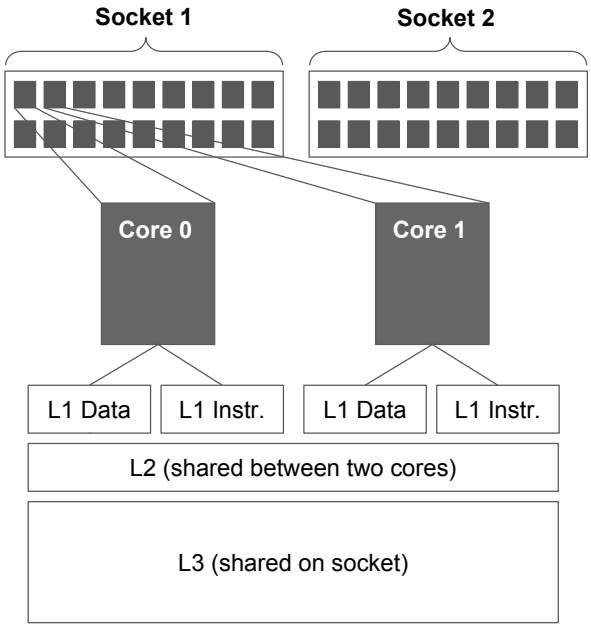


Figure 5.1: Test platform architecture.

the wide gap between the lock overhead for the U-C-RNLP compared to that for MCS. The objective in this chapter is to narrow this gap, hopefully considerably. Many such graphs are carefully examined in Section 5.4, and additional descriptions are given there.

Lock servers. Recall that the focus of this section is static lock servers that are pinned to dedicated cores. Two variations of this idea are considered: using a *global* lock server that services requests from all cores, and using (on the test platform) two *local* lock servers, each servicing requests coming from one socket. Figure 5.3 depicts these two possibilities in comparison to a conventional locking protocol implementation that does not use lock servers. The potential value of lock servers can be seen by comparing the curve for U-C-RNLP + SGLS to the U-C-RNLP curve in Figure 5.2. (Again, graphs like this are considered in detail later.)

5.1.1 A Static Global Lock Server

The simplest way to employ a lock server is to dedicate a single core to servicing all lock requests. The server uses a special version of a given protocol's LOCK call, denoted LS-LOCK, that updates the lock state to add a given request and then, instead of waiting by spinning to be satisfied, returns the location of a variable

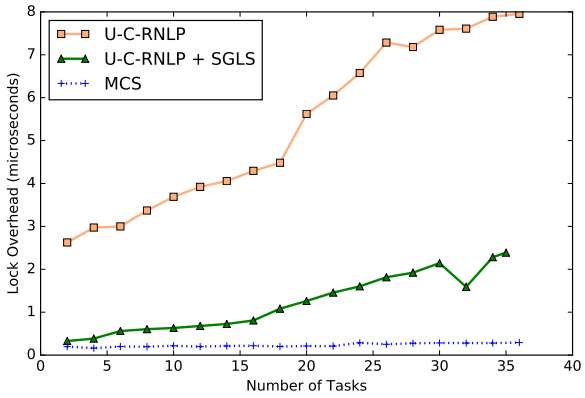


Figure 5.2: Lock overhead under the U-C-RNLP with and without a lock server.

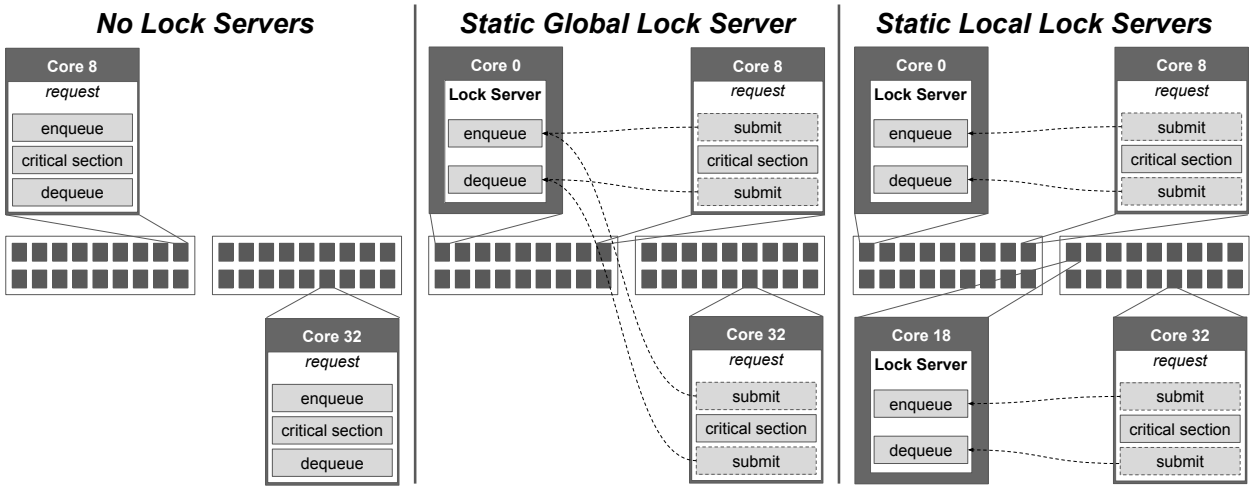


Figure 5.3: Three options: no lock servers (left), a single static global lock server (middle), and two per-socket static local lock servers (right).

on which to spin. Similarly, a special version of UNLOCK, denoted LS-UNLOCK, is used. Note that these routines require no underlying mutex, as no task other than the lock server will ever access the lock state.

The behavior of the lock server is as specified in Listing 8. It is continually active (Line 3), looping through each core (Line 10). By limiting focus to non-preemptive, spin-based protocols, it is guaranteed that each core will have at most one active request at a given time. For a specific core k, the server checks if there is an active request that needs lock service (Line 4). If so, it uses LS-LOCK to add the request to the lock state and determine the spin location for it (Line 5). In the case of the U-C-RNLP, this is the entry in *Enabled* that corresponds to the row in *Table* to which the request was added. The server then indicates that this core no longer requires service (Line 6). If instead, a request on core k requires unlock service (Line 7),

Listing 8 Static Global Lock Server

```
 1: procedure SGLS(core: array of ptr to core_data)
 2:     var k: unsigned int
 3:     while (TRUE):
 4:         if core[k]→service = LOCK_SERVICE:
 5:             ^core[k]→spin_location := LS-LOCK(core[k]→requested)      ▷ Non-blocking LS-LOCK returns spin location
 6:             core[k]→service := NULL
 7:         else if core[k]→service = UNLOCK_SERVICE:
 8:             LS-UNLOCK(core[k]→requested)
 9:             core[k]→service := NULL
10:         k := k + 1 mod NR_CPUS
11: end procedure
```

Listing 9 New "Lock" and "Unlock" Submit Routines

```
 1: procedure SUBMIT-LOCK(c: ptr to core_data, D: set of resources)
 2:     c→requested := D
 3:     c→service := LOCK_SERVICE
 4:     await c→service = NULL
 5:     await c→spin_location = TRUE
 6: end procedure
 6: procedure SUBMIT-UNLOCK(c: ptr to core_data, D: set of resources)
 7:     c→requested := D
 8:     c→service := UNLOCK_SERVICE
 9:     await c→service = NULL
10: end procedure
```

the server removes it from the lock state by calling LS-UNLOCK (Line 8). It then updates the service variable indicating that core k no longer requires service (Line 9).

The next example illustrates the behavior of a requesting task.

Example 5.1. Figure 5.4 shows the result of processing a request $\mathcal{R}_5$ for $D_5 = \{\ell_a, \ell_b\}$ that is issued after requests $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$, and $\mathcal{R}_4$. With a single global lock server, $\mathcal{R}_5$ executes SUBMIT-LOCK as shown in Listing 9. It first sets *Requested* (Line 2) for its core and then indicates that it is awaiting lock service by the server (Line 3). After it has been serviced (Line 4), it spins on the location the server determined based on the other active requests (Line 5). As implied by Figure 5.4, $\mathcal{R}_5$ spins on *Enabled*[3]. ◊

Using a global lock server in this manner has no impact on blocking; it simply changes the enqueuing and dequeuing portions of request processing in order to reduce overhead.

5.1.2 Static Local Lock Servers

In contrast to a global lock server, a local one is allowed to handle resource requests from only one socket. The test platform has two sockets, so two lock servers are required to handle all requests; they are denoted as $\mathcal{LS}_1$ and $\mathcal{LS}_2$. In this section, the focus is on static lock servers, which means that each lock

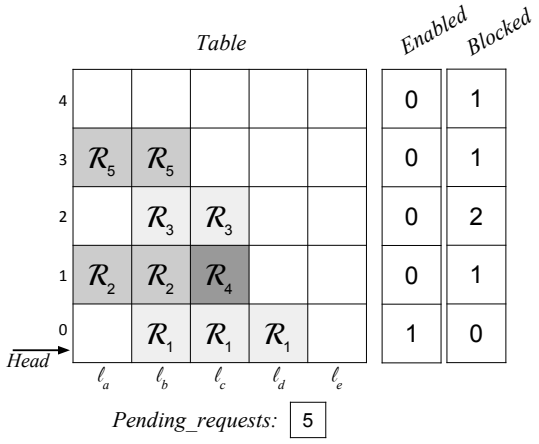


Figure 5.4: $\mathcal{R}_5$ is added to Row 3 of *Table*.

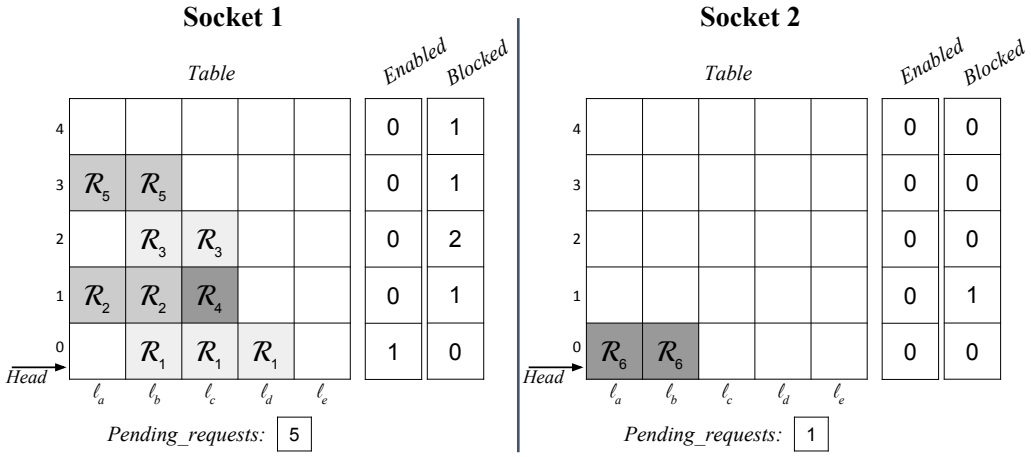


Figure 5.5: $\mathcal{R}_6$ is added to *Table* of Socket 2.

server is pinned to a specific core on its socket. The advantage of having two lock servers is that each must handle requests from only half the cores, and thus should execute with lower overhead. The disadvantage is that some arbitration mechanism is needed to mediate conflicting requests managed by the two servers. The nature of the needed mediation is illustrated with an example.

Example 5.1 (continued). Suppose that the requests in Figure 5.4 were actually issued on Socket 1. Suppose now a request $\mathcal{R}_6$ for $D_6 = \{\ell_a, \ell_b\}$ is issued on Socket 2. This results in the two lock states shown in Figure 5.5. Though $\mathcal{R}_6$ is the only request in $\mathcal{LS}_2$'s lock state, it should not be satisfied, as it conflicts with request $\mathcal{R}_1$ for resource ℓ_b. Thus, it must wait. $\Diamond$

Mediating requests from the two lock servers can be accomplished by allowing them to alternate execution in phases. Section 5.5 presents a phase-management protocol to coordinate these phases. In the U-C-RNLP, a

Listing 10 Static Local Lock Server

```
1: procedure SLLS(core: array of ptr to core_data, s: socket identifier)
2:     Service lock and unlock requests like in Alg. 1, but with the following changes:
3:         Only requests from the local socket s are handled
4:         Coordinate Phase with other lock server
5:         Set spin_location := TRUE for requests that are eligible to be satisfied while Phase = s
6: end procedure
```

natural way to define which requests belong to a certain phase is to let each row of *Table* indicate a phase. As shown in Section 5.5, when defining and managing phases in this way, the blocking experienced by request $\mathcal{R}_i$ is at most $(c_{i,s}+1)(L_{max,1}+L_{max,2})$ time units, where $c_{i,s}$ is the contention $\mathcal{R}_i$ experiences on Socket s and $L_{max,s}$ is the maximum critical-section length on Socket s. In Listing 10, this boundary and change between phases is coordinated in Line 4 and the current phase is stored in the variable *Phase*. The coordination must ensure bounded time before a change of *Phase* when requests are waiting on the other socket. Thus, in Line 5, a request must be *able* to be satisfied (*e.g.*, it is in the active row of *Table* in the U-C-RNLP) and the phase must be set to the local socket before the request can be *marked* as satisfied by updating its spin location.

5.2 Floating Lock Servers

The prior section implicitly assumed that static lock server(s) are to be supported by devoting full core(s) to them. While this may be reasonable on a large platform, it would be possible to instead allow other work to execute on the core(s) assigned to static lock servers(s) as long as that work executes at a lower priority. The impact lock servers have on such work could be assessed similarly to how interrupt accounting is done.

This section explores a simpler alternative: floating lock servers. When using static lock servers, every request executes a spin loop for each server interaction in order to wait for a response. When using floating lock servers, the processor time wasted during these spin loops is reclaimed to execute lock-server code. This approach is tantamount to employing a *helping mechanism* (Herlihy, 1991), but unlike the traditional sense of helping, where one request may help another to complete a *critical section*, a request here performs only *lock logic* on behalf of another request. The floating lock-server paradigm is described more fully below by first considering global servers and then local ones.

Listing 11 Floating Global/Local Lock Server

```
 1: global var Server_exists: boolean initially FALSE

 2: procedure FLOATING-LOCK(c: ptr to core_data, D: set of resources)
 3:     var i_am_server: boolean initially FALSE
 4:     c→requested := D
 5:     c→service := LOCK_SERVICE
 6:     i_am_server := WAIT-UNTIL(^(c→service), NULL)
 7:     if (i_am_server = FALSE):
 8:         i_am_server := WAIT-UNTIL(^(c→spin_location), TRUE)
 9:     if (i_am_server = TRUE):
10:         while (c→service ≠ NULL) or (c→spin_location ≠ TRUE):        ▷ Until satisfied, be server
11:             Perform lock server functionality
12:         Server_exists := FALSE
13: end procedure

14: procedure FLOATING-UNLOCK(c: ptr to core_data, D: set of resources)
15:     var i_am_server: boolean initially FALSE
16:     c→requested := D
17:     c→service := UNLOCK_SERVICE
18:     i_am_server := WAIT-UNTIL(^(c→service), NULL)
19:     if (i_am_server = TRUE):
20:         if c→service ≠ NULL:                                        ▷ This request has not been serviced
21:             Perform unlock for this request
22:         Server_exists := FALSE
23: end procedure

24: procedure WAIT-UNTIL(location: ptr, value)
25:     var t: unsigned int
26:     t := TestAndSet(&Server_exists)
27:     while (t = TRUE) and (*location ≠ value):
28:         if (Server_exists = FALSE):
29:             t := TestAndSet(&Server_exists)                         ▷ TestAndSet return value of FALSE means ...
30:     return (t = FALSE)                                              ▷ ... Server_exists was FALSE so I am now server
31: end procedure
```

5.2.1 A Floating Global Lock Server

This section more carefully describes the notion of a floating global lock server. Unlike static lock servers, in floating ones, request code and lock-server code are inextricably linked. Thus, how a floating global lock server works is specified via one code listing in Listing 11.

In Listing 11, a request in its lock call performs the same logic as it would using a static server (marking itself as requiring service, waiting for a location on which to spin, and then spinning), with intermediate checks to ensure that some request is acting as the lock server. The existence of a lock server is maintained in the global variable *Server_exists*. The helper method WAIT-UNTIL waits until a designated location holds a desired value, with the waiting terminated if the caller becomes the server (as determined in a test-and-test-and-set manner). The return value of this method indicates whether the caller is now the server.

Examining the FLOATING-LOCK routine in a bit more detail, a request first marks that it is ready to be serviced (Line 5). Then, it waits to be serviced (Line 6). If it is not the lock server, then it spins on *spin_location* (Line 8). If it becomes the lock server, then it performs the lock server functionality until it is satisfied (Lines 10-11). Notice that whenever a request functions as the lock server here, it would have been spinning in the global static lock server paradigm waiting for a server response.

The FLOATING-UNLOCK routine is similar, except that a request that becomes the lock server only services itself (Line 21). This is because an unlock does not involve blocking, so servicing other requests would not replace useless spinning, but would just slow the unlock.

5.2.2 Floating Local Lock Servers

While a floating global lock server has the benefit over static lock server(s) of not requiring dedicated core(s), it also would be expected to suffer higher overhead due to eroded cache affinity when lock state moves between sockets. Fortunately, there is a quick fix to keep lock state in cache: implement a floating *local* lock server. In this paradigm, a request can only perform the functions of the lock server for the socket from which it was issued. By restricting to a single socket, L3 cache affinity can be maintained. A floating local lock server uses the structure found in Listing 11, but with the server logic in Lines 11 and 21 being that of a local lock server (with phase arbitration).

5.3 Handling Non-Uniform Requests

Recall from Section 5.1 that the C-RNLP is defined in an abstract rule-based way and that the U-C-RNLP is just one implementation of it. The U-C-RNLP can be used to handle non-uniform requests by pessimistically viewing all critical sections as L_{max}. However, this changes the worst-case blocking bound of the general version from $\min(mL_{max}, c_i(L_{max}+L_i))$ to $\min(m,(c_i+1))L_{max}$ (Jarrett et al., 2015). This section discusses an alternate non-uniform implementation, denoted as the G-C-RNLP, that maintains the original bound.

The G-C-RNLP uses $|D_i|$ nodes to represent a request $\mathcal{R}_i$, one corresponding to each resource in D_i. A separate queue is maintained for each resource in the system. When $\mathcal{R}_i$ is processed, a satisfaction time is recorded for it by considering the satisfaction times for other requests and the critical-section length of each. Then, the queue for each resource in D_i is updated by inserting a node for $\mathcal{R}_i$ at a position that ensures that

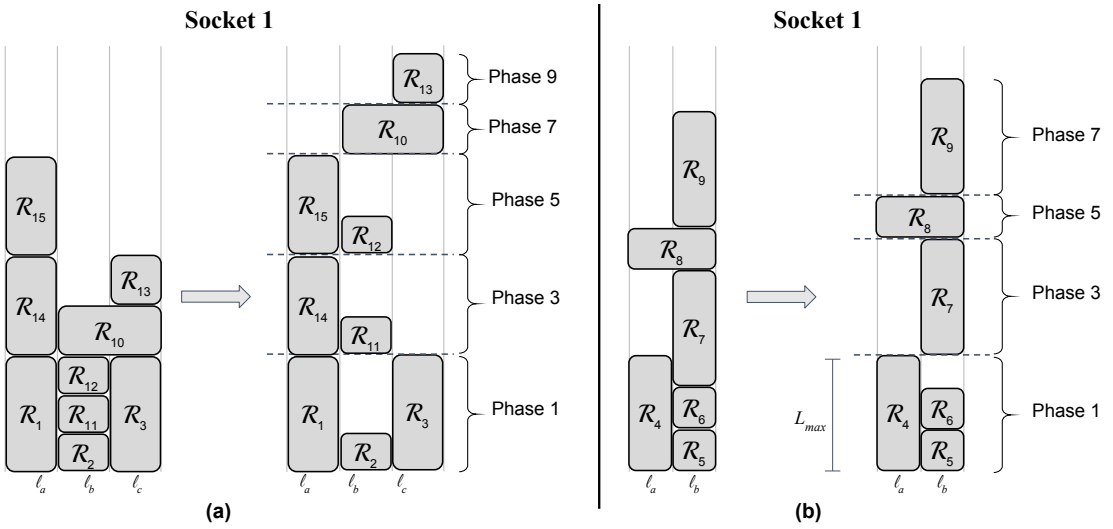


Figure 5.6: Scenarios with complicated phase management.

$\mathcal{R}_i$ will be at the head of its respective queues by its recorded satisfaction time. This protocol gives rise to prohibitively high overhead if the tasks themselves execute the queuing logic concurrently. In particular, when enqueuing a request $\mathcal{R}_i$, $|D_i|$ queues must be checked for the satisfaction times of existing requests, and $|D_i|$ nodes must be inserted (sometimes in the middle of queues). However, if this protocol is implemented using lock servers,[2] then the overhead becomes quite reasonable, as shown in Section 5.4.

Using global lock servers (Sections 5.1.1 and 5.2.1) to implement the G-C-RNLP is straightforward: it merely requires using the G-C-RNLP instead of the U-C-RNLP in the LS-LOCK and LS-UNLOCK routines. On the other hand, using local lock servers (Sections 5.1.2 and 5.2.2) is more problematic due to the phase management such servers require. This problem is illustrated in the following two examples. For the time being, assume that a basic phase-management protocol called *Greedy Satisfaction* is used that allows only requests that can be satisfied at the start of a phase to be satisfied during that phase.

Example 5.2. Consider the requests shown in Figure 5.6 (a), all issued on Socket 1. $\mathcal{R}_2$, $\mathcal{R}_{11}$, and $\mathcal{R}_{12}$ are "short" requests for resource ℓ_b and most of the other requests (for various resources) are longer. Under Greedy Satisfaction, requests would be satisfied in phases as shown in the right half of Figure 5.6 (a), with dashed lines indicating phase boundaries. Observe that, under this policy, only $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_3$ are satisfied in the first phase. $\mathcal{R}_{11}$ and $\mathcal{R}_{12}$ are satisfied later. Notice that all of the phases have odd indicies. This is because Socket 2 executes during even-indexed phases. ◊

[2] Although not reflected in the pseudocode given here, the lock-server implementations have been carefully honed using bit-vector operations and other techniques to improve efficiency.

Example 5.2 shows that Greedy Satisfaction can unnecessarily delay requests: $\mathcal{R}_{11}$ and $\mathcal{R}_{12}$ both could have completed by the time $\mathcal{R}_3$ completed. Instead, they are moved to two later phases. This is called the *Long-Short Problem*: when requests vary in length, shorter requests can be delayed, further delaying other requests. In this example, $\mathcal{R}_{13}$ in particular is delayed substantially by requests with which it does not conflict.

Example 5.2 highlights the fact that, for some protocols, Greedy Satisfaction is inadequate. A better solution is a policy called *Timed Satisfaction*, which allows requests that can finish within L_{max} time units to be satisfied in the same phase.

Example 5.3. In Figure 5.6 (b), Timed Satisfaction is applied to a different set of requests on Socket 1. On the left, the requests are shown as they are ordered by the G-C-RNLP. On the right, the requests are shifted to occupy the phases the lock server would enforce. $\mathcal{R}_4$ and $\mathcal{R}_5$ are satisfied at the start of Phase 1. After $\mathcal{R}_5$ completes, $\mathcal{R}_6$ is also satisfied in this phase. However, after $\mathcal{R}_6$ completes, $\mathcal{R}_7$ cannot be satisfied, as it cannot be guaranteed to complete within L_{max} time units from the start of the phase. Therefore, $\mathcal{R}_7$ must wait until Socket 2 is allowed another phase, namely, Phase 3. ◊

Example 5.3 illustrates another source of added blocking: $\mathcal{R}_7$ is forced to delay until the start of the next phase to be satisfied. Even if the time window were increased, the problem could arise again: another request could be issued that cannot complete within the window. This difficulty is called the *Overlap Problem*. A phase must end at some point to prevent the starvation of requests on the other socket. Whatever value is chosen, requests may overlap a phase boundary and need to be delayed. The Overlap Problem can force a request that could otherwise be satisfied to be delayed until the current phase of its lock server completes followed by a full phase of the other lock server before being satisfied.

When considering the effect of local lock servers on blocking with the G-C-RNLP, Timed Satisfaction was assumed as the phase-management policy. (Again, the issues just discussed are unique to local servers.) As seen in Example 5.3, Timed Satisfaction is susceptible to the Overlap Problem. This is the reason why the worst-case blocking bounds presented in Section 5.5 for the G-C-RNLP are worse than those for the U-C-RNLP.

5.4 Evaluation

The primary reason for exploring lock servers is to minimize overhead by keeping all lock state cache hot. For static global and static local servers, cache hot means the lock state should maintain L1 cache affinity on the platform, whereas a floating local server should tend to execute out of its socket's L3 cache. On the other hand, a floating global server will likely not be able to maintain much cache affinity if tasks execute on more than one socket.

Given these expectations, a number of questions arise. How do the different lock-server paradigms presented previously differ with respect to overhead, and do these differences match the above expectations? To what extent do lock servers lower overhead compared to not using lock servers? Are the overhead improvements enough to make contention-sensitive locking practical? How do lock servers scale with increasing core counts?

An experimental study was designed to answer these questions. Before covering the results revealed by this study, the experimental setup is described.

5.4.1 Experimental Setup

Recall from Section 5.1 that the considered test platform is a dual-socket, 18-cores-per-socket platform. This platform was used to evaluate the lock-server paradigms discussed previously by conducting experiments involving tasks that repeatedly issue lock and unlock calls for random resources. A wide range of parameters were varied to evaluate each parameter's effect on overhead and blocking: the number of tasks, n, number of resources, n_r, nesting depth (which defines the number of resources required for request R_i), $\mathbb{D} = |D_i|$, and critical-section length, L_i. A *scenario* is an assignment of values to three of these parameters while varying the fourth. The following parameter ranges were considered: $n \in \{2, 4, ..., 36\}$, $n_r \in \{16, 32, 64\}$, $\mathbb{D} \in \{1, 2, 4, ..., 10\}$, and $L_i \in \{1\mu s, 20\mu s, 40\mu s, ..., 100\mu s\}$. In the experiments, all requests in a scenario have the same nesting depth. Unless stated otherwise, they also all have the same critical-section length L_i.

Overhead and blocking times were recorded at user level, with one task pinned to each core. This setup ensures that requests execute non-preemptively. For a given scenario, each task to performed 10,000 lock and unlock calls, with critical sections simulated by spinning for a duration of L_i. For task systems running on at most 18 cores, only the cores on one socket were used. When using more than 18 cores, all cores on Socket 1 were used with the remainder on Socket 2. The workload is comprised solely of tasks making lock

and unlock calls as described above. Thus, this evaluation focuses on cache affinity losses inherent to running a protocol and ignores potential evictions from other tasks; there exist techniques to keep cache affinity in some systems (Altmeyer et al., 2014; Campoy et al., 2001; Chisholm et al., 2015; Herter et al., 2011; Kim et al., 2013; Kirk and Strosnider, 1990; Ward et al., 2013; Xu et al., 2016, 2017; Yun et al., 2014).

In the graphs that follow, the 99$^{\text{th}}$ percentile measurements are plotted as worst-case values to filter out any spurious measurements caused by performing measurements at user level.[3] Across over 150 scenarios, this generated approximately 1,000 graphs. The graphs shown in this section were chosen as examples of trends seen across the entire collection of graphs.[4]

5.4.2 Overhead and Blocking without Lock Servers

Before delving into results pertaining to lock-server paradigms, let us examine a range of server-less implementation options. To gauge the tradeoffs involved in supporting lock nesting, this experimental evaluation includes two contention-sensitive options, the U-C-RNLP and the G-C-RNLP, both implemented without lock servers, and the RNLP, which supports nesting but is not contention sensitive. As a baseline, coalescing all resources under one MCS queue lock (Mellor-Crummey and Scott, 1991a) is also evaluated. These options were compared on the basis of blocking and overhead.

The subsequent observations follow from the full range of scenarios considered in these experiments. Each observation is illustrated using the graphs in Figure 5.7.[5] In this figure, lock and unlock overhead are presented separately to demonstrate their relative scale: enqueuing takes slightly longer than dequeuing, but both operations require manipulating lock state, and thus both contribute to overhead. In later figures, lock and unlock overheads are combined to yield one overhead graph.

Observation 5.1. Without using lock servers, both C-RNLP variants had dramatically higher overhead than MCS.

This is expected behavior, as MCS implements just a single spin queue. As shown in insets (b) and (c) of Figure 5.7, the U-C-RNLP had lock and unlock overhead up to 27.4 and 23.9 times that of MCS, respectively. For the G-C-RNLP, these values were similarly high: up to 31.1 and 22.9 times, respectively.

[3]This filtering does not guarantee smoothness of all curves.

[4]The full set of plots and the source code are available at `https://www.cs.unc.edu/~jarretc/dissertation/`.

[5]In every such figure, the applicable scenario is stated in the figure's caption. Note that not all curves extend to $n = 36$. This is because up to two cores are reserved for lock servers and this number is scheme-dependent.

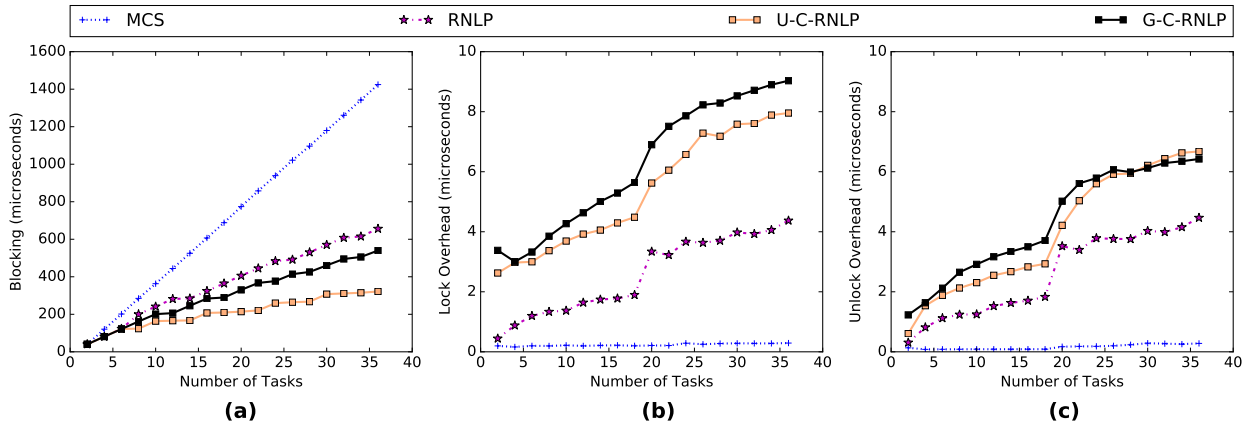


Figure 5.7: Blocking and lock/unlock overhead when no lock servers are used. For this scenario, $n_r = 64, \mathbb{D} = 4$, and $L_i = 40\mu s$ for all i.

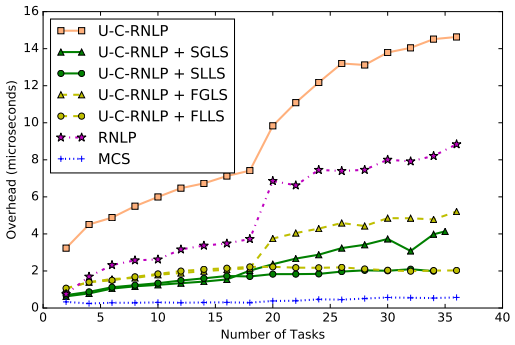
Observation 5.2. Compared to MCS, contention-sensitive protocols demonstrated significantly better blocking bounds as the number of requests increases.

The low overhead of MCS (Observation 5.1) comes at the expense of unscalable blocking. As shown in Figure 5.7 (a), worst-case blocking under MCS grew up to 5.3 and 2.9 times faster than that under the U-C-RNLP and G-C-RNLP, respectively.

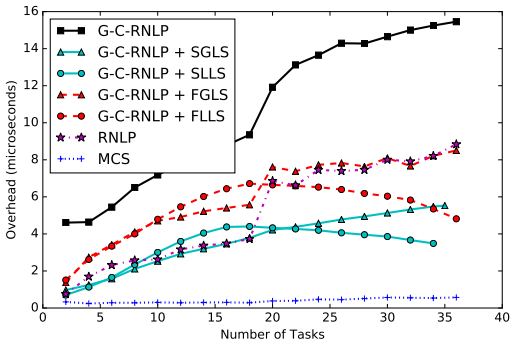
Considering the RNLP is instructive because it provides some insights into the extra cost of providing contention sensitivity in addition to handling lock nesting. As shown in insets (b) and (c) of Figure 5.7, lock and unlock overhead under the U-C-RNLP (respectively, G-C-RNLP) were up to 1.8 and 2.1 (respectively, 1.5 and 1.4) times that under the RNLP, respectively.

5.4.3 Applying Lock Servers

Sections 5.1 and 5.2 presented four lock-server paradigms, each of which can be applied to any locking protocol. Experiments were conducted to explore how these paradigms differ when used to implement the U-C-RNLP and the G-C-RNLP. The subsequent observations follow from the full range of scenarios considered in these experiments. These observations are illustrated using the graphs in Figures 5.8 and 5.9. In Figure 5.8 (a), the four possible lock-server variants of the U-C-RNLP are compared against the baselines of MCS, the RNLP, and the U-C-RNLP without lock servers. Figure 5.8 (b) is similar, but is directed at the G-C-RNLP instead of the U-C-RNLP. (Lock-server paradigms are abbreviated in figure captions, *e.g.*, static global lock server is SGLS.)



(a) U-C-RNLP overhead



(b) G-C-RNLP overhead

Figure 5.8: For this scenario, $n_r = 64, \mathbb{D} = 4$, and $L_i = 40\mu s$ for all i.

Observation 5.3. Using lock server(s) resulted in significantly lower overhead.

This can be seen both in Figure 5.8 (a) for the U-C-RNLP and in Figure 5.8 (b) for the G-C-RNLP. Observe that using lock server(s) usually resulted in overhead even lower than that of the RNLP. In fact, using local lock servers in this scenario reduced the overhead of the U-C-RNLP and the G-C-RNLP by up to 86% and 77%, respectively.

Observation 5.4. When there were requests on only one socket, static lock servers resulted in the largest overhead reduction.

This trend appeared consistently across the results, and matches our intuition, as a static lock server can maintain L1 cache affinity. In Figure 5.8, only one socket was used when $n < 18$ (it is strictly less because the lock server uses one core).

Observation 5.5. When considering requests on two sockets, as the number of tasks increased, the overhead of local lock servers scaled better than that of a global lock server.

For example, in Figure 5.8, the overhead of the U-C-RNLP (respectively, G-C-RNLP) with floating local lock servers was up to 61% (respectively, 43%) lower than with a floating global lock server.

Observation 5.6. Floating global lock servers scaled the poorest of the four lock-server paradigms.

This observation was entirely expected and clearly evident in Figure 5.8. Note that a floating global lock server still reduced overhead to be comparable to or better than the RNLP.

In Figure 5.9, worst-case blocking under the U-C-RNLP is plotted for each lock-server paradigm for the same scenario presented in Figure 5.8.

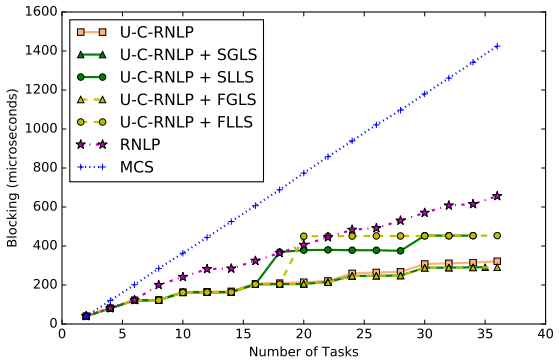


Figure 5.9: Worst-case blocking for the scenario in Figure 5.8 (a).

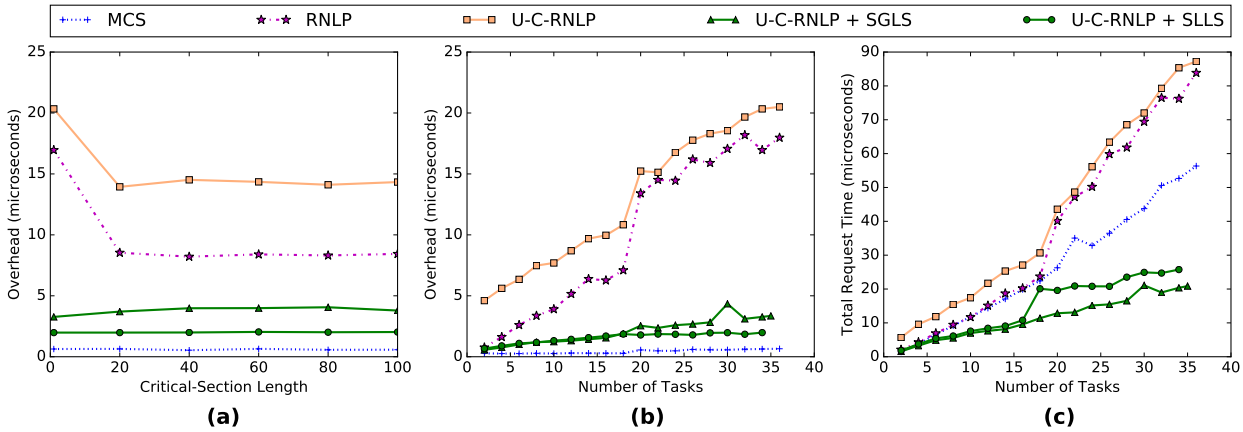
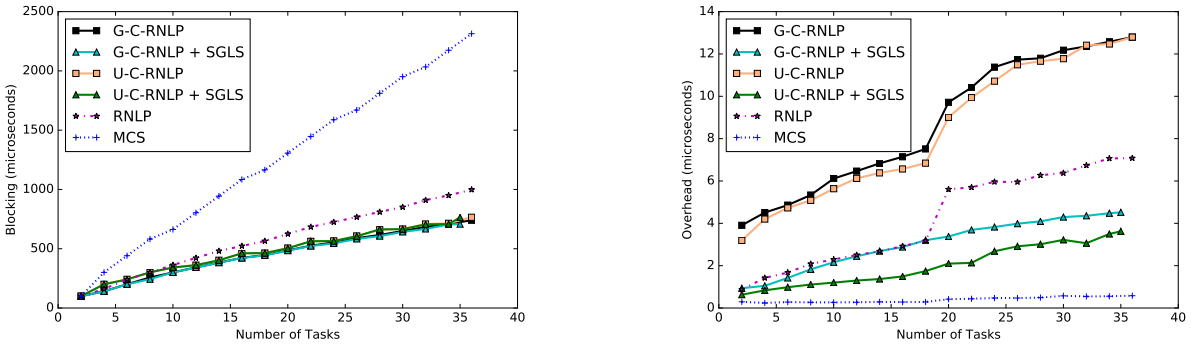


Figure 5.10: **(a)** Overhead as a function of critical-section length, for $n = 34, n_r = 64$, and $\mathbb{D} = 4$. **(b)** Overhead and **(c)** blocking as a function of n, for $n_r = 64, \mathbb{D} = 4$, and $L_i = 1\mu s$ for all i.

Observation 5.7. Moving from one socket to two can negatively impact blocking of local lock servers.

This observation is evident in Figure 5.9. Two local lock servers are required if $n \geq 18$. The extra blocking is due to phase management and request imbalances between the two sockets. For example, for $n = 18$, there were 17 requests on Socket 1 and one request on Socket 2. The request on Socket 2 would have very low blocking, but requests on Socket 1 could experience twice as much blocking as when only one socket was in use. Without the mitigation described in Section 5.5, blocking scaled poorly with increasing socket counts (revisited in Section 5.4.4).

Requests with short critical sections. Inset (a) of Figure 5.10 plots overhead as a function of critical-section length, while insets (b) and (c) provide data for a scenario with a short critical section of $1\mu s$. (The G-C-RNLP variants are omitted from this figure for clarity; overhead for them was higher than their



(a) Blocking

(b) Overhead

Figure 5.11: For this scenario, $n_r = 64, \mathbb{D} = 4$, and $L_i = 40\mu s$ for 75% of requests and $L_i = 100\mu s$ for the remaining 25% of requests.

U-C-RNLP counterparts but followed similar trends.) Such short critical sections result in overhead being a higher proportion of total request time (overhead plus blocking). Note that the blocking time of a request includes the overhead of any request upon which it must wait, so reducing overhead additionally reduces blocking.

Observation 5.8. Overhead was (mostly) constant for all U-C-RNLP variants with respect to L_i.

This is demonstrated in Figure 5.10 (a). Note that, when static lock servers are used, overhead remained low even for small L_i.

Observation 5.9. When critical sections were short, lock servers greatly reduced the impact of overhead on total request time.

The data in insets (b) and (c) of Figure 5.10 indicates that, under the U-C-RNLP, requests with $1\mu s$ critical sections can experience worst-case overhead that is up to 23.4% of the total request time. When using a static local lock server, this was reduced to 9.6%.

A case where the G-C-RNLP wins. From the results presented thus far, it is tempting to discount the G-C-RNLP entirely. In cases where all critical sections were of the same duration, the G-C-RNLP suffered worse overhead and blocking than the U-C-RNLP. Now let us explore scenarios in which the G-C-RNLP had very competitive worst-case blocking; this occurred when a large fraction of requests have critical-section lengths much less than L_{max}. Such a scenario is depicted in Figure 5.11.

	U-C-RNLP	U-C-RNLP + SGLS	U-C-RNLP + SLLS	U-C-RNLP + FGLS	G-C-RNLP + SGLS
Total Firsts	0	92	0	23	12
Total Seconds	1	26	18	70	4
Total Thirds	68	2	17	20	8
Total	69	120	35	113	24

Figure 5.12: Results of total request time comparison.

Observation 5.10. When most requests had critical sections much shorter than L_{max}, the G-C-RNLP and U-C-RNLP had similar performance when both used a static global lock server.

In Figure 5.11, the G-C-RNLP with a static global lock server has lower blocking and only slightly higher overhead than the U-C-RNLP with the same lock-server setup.

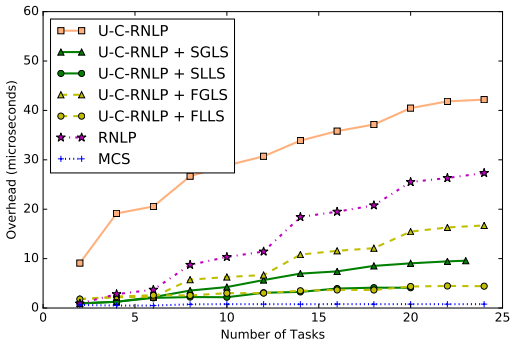
Overall winner. Judging the lock-server paradigms should be done with a specific workload, but to make a general summary, the "best" paradigm was determined to the extent possible in this experimental framework as follows. For each considered scenario,[6] a single "total request time" score (blocking plus overhead) was calculated for each protocol variant by approximating the area under its curve using a midpoint Riemann sum. The protocol variants are then ranked for that scenario. Figure 5.12 gives the total number of first-, second-, and third-place finishes for each protocol variant. The U-C-RNLP with a static global lock server was the overall winner. However, the experimental setup mostly generates scenarios in which critical sections are uniform, which tends to make the G-C-RNLP variants less competitive. Still, these results show there is value in using lock servers.

5.4.4 Results on an Alternate Platform

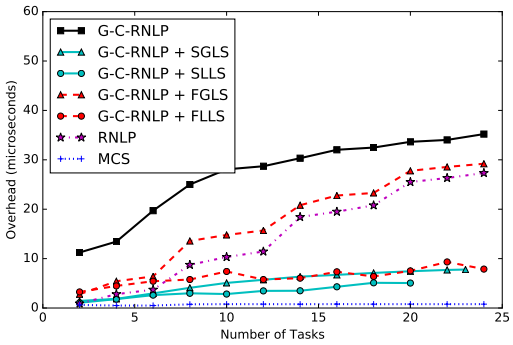
Recall from Section 5.1 that the original test platform is a dual-socket, 18-cores-per-socket platform. In order to examine how these approaches scale with increasing socket count, the experiments in Figures 5.8 and 5.9 were repeated on an alternate experimental platform.

This alternate test platform is a four-socket, 6-cores-per-socket Intel Xeon L7455. On this machine, each core has a 32KB L1 data cache and a 32KB L1 instruction cache. Additionally, there is a 3MB L2 cache, and all cores on a socket share a 12MB L3 cache.

[6] Scenarios with $\mathbb{D} \in \{8, 10\}$ were filtered out, as they require nearly coalescing all resources under a single lock, which has non-contention-sensitive blocking.



(a) U-C-RNLP overhead



(b) G-C-RNLP overhead

Figure 5.13: Same scenario as in Figure 5.8: $n_r = 64, \mathbb{D} = 4$, and $L_i = 40\mu s$ for all i.

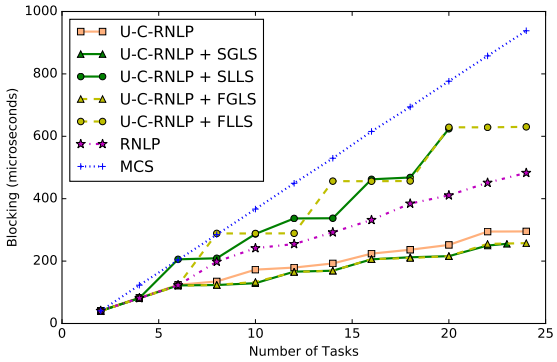


Figure 5.14: Worst-case blocking for the scenario in Figure 5.13 (a).

Section 5.4 presented the results of using lock servers on up to two sockets. The experiments depicted in Figures 5.8 and 5.9 were repeated on the alternate platform; these results are shown in Figures 5.13 and 5.14 for up to four sockets.

These figures validate Observations 5.3–5.6 on up to four sockets. Additionally, Observation 5.7 can be extended to increasing socket counts, given by the following observation.

Observation 5.11. Blocking of local lock servers scaled poorly with increasing socket counts.

This observation is supported by Figure 5.14. In this case, for $n \in [6, 12)$, two sockets were required. Similarly for $n \in [12, 18)$ and $n \geq 18$, three and four sockets were used, respectively. In this scenario, the requests were not balanced between the four sockets, so for $n = 6$, Socket 1 had five requests and Socket 2 had the last one. As before, the mitigation in Section 5.5 was not used here, so the blocking scaled poorly as the number of sockets continued to increase, to the point of being worse than that of the RNLP.

5.5 Local Lock Server Phase Management and Blocking Bounds

This section provides additional details concerning the phase-management protocol needed for the local lock servers described in Sections 5.1.2 and 5.2.2. A local lock server must determine which requests will execute in each of its phases in addition to managing phase changes.

Request selection. Each phase on Socket s is restricted to executing for at most the maximum critical-section length on that socket, denoted $L_{max,s}$. For the U-C-RNLP, the requests in a phase are determined by selecting the row in *Table* pointed to by *Head*. For the G-C-RNLP, Timed Satisfaction (recall Section 5.2.2) is used instead.

Phase coordination. Because all requests that can be satisfied simultaneously under C-RNLP rules can run concurrently relative to each other, they may be processed like read requests. With this in mind, the synchronization mechanism needed can be obtained by building on the idea of a phase-fair reader/writer lock (Brandenburg and Anderson, 2010). Recall from Section 2.6.2.3 that in this locking protocol, read and write phases alternate if both kinds of requests are present, and any number of reads can occur during a read phase. The synchronization mechanism desired for lock servers similarly needs to support two kinds of requests that execute in alternating phases, but in this case, any number of requests can execute in a given phase. That is, a *reader/reader* lock is needed. This section presents a new phase-fair reader/reader locking algorithm with corresponding blocking bounds. (Recall from Section 3.1.1 that the phase-fair reader/reader problem is similar to the group mutual exclusion problem (Joung, 2000; Keane and Moir, 1999, 2001) except that the phase-fair designation requires $O(1)$ pi-blocking bounds.)

Phase management can be supported by a reader/reader protocol that adheres to the following rules. Recall from Section 5.1.2 that $\mathcal{LS}_s$ denotes the local lock server on Socket s and that $L_{max,s}$ denotes the maximum critical-section length on Socket s.

S1 Each lock server is either *active* or *passive* and at most one lock server is active at any time. A maximal interval of time when a lock server is active is called a *phase*.

S2 A request can be satisfied only if its lock server is active and if it can be satisfied under the variant of the C-RNLP being used by that server.

S3 A passive lock server $\mathcal{LS}_1$ (respectively, $\mathcal{LS}_2$) with unsatisfied requests becomes active within $L_{max,2}$ (respectively, $L_{max,1}$) time units.

S4 All requests satisfied in a phase finish before the end of that phase.

S5 When the last request of a phase finishes, the completion of that request and the transition to a new active phase happens atomically.

In the remainder of this section, bounds on worst-case acquisition delay are presented. In stating these bounds, the contention a request $\mathcal{R}_i$ experiences on Socket s is denoted $c_{i,s}$. A request on Socket s is *entitled* if it could be satisfied under the C-RNLP variant used by $\mathcal{LS}_s$. In the U-C-RNLP, a request is entitled when it is in the row pointed to by *Head*. Exactly one row is satisfied in each phase, as discussed above. For the G-C-RNLP, a request is entitled if it is the head of the queue for all of its required resources. Here it is assumed that the G-C-RNLP uses *Timed Satisfaction* as its phase management policy.

Lemma 5.1. *Under the U-C-RNLP, a request $\mathcal{R}_i$ handled by server $\mathcal{LS}_1$ (respectively, $\mathcal{LS}_2$) becomes satisfied within $L_{max,2}$ (respectively, $L_{max,1}$) time units after becoming entitled.*

Proof. Let us begin by showing the bounds for request $\mathcal{R}_i$ on Socket 1. Consider the time instance t when $\mathcal{R}_i$ becomes entitled. At t, $\mathcal{LS}_1$ is either active or passive (Rule S1). If $\mathcal{LS}_1$ is passive, it will become active within $L_{max,2}$ time units, at which point $\mathcal{R}_i$ would be satisfied (Rules S2 and S4). If instead $\mathcal{LS}_1$ is active, then it must have become active at t. (Under the U-C-RNLP, requests are never added to the row pointed to by *Head* when other requests are active, as discussed in Chapter 4. Also, the row does not change in the middle of a phase when there are requests on the other socket (Rule S5).) Thus, t is the start of a phase, and $\mathcal{R}_i$ can complete within L_{max} time units because $L_i \leq L_{max}$ by definition.

The proof for $\mathcal{R}_i$ on $\mathcal{LS}_2$ follows the same pattern. □

Example 5.4. Consider a request $\mathcal{R}_1$ on Socket 2 for $D_1 = \{\ell_a\}$ that is satisfied at time t. $\mathcal{R}_2$ on Socket 1 for $D_2 = \{\ell_a\}$ issued at time $t + \varepsilon$ is entitled; it is the only active request on Socket 1 and is in the row pointed to by *Head*. It will be satisfied when $\mathcal{R}_1$ completes, which will occur at time $t + L_{max,2}$ at the latest. ◊

Theorem 5.11. *A request $\mathcal{R}_i$ on Socket s that is serviced by a local lock server running the U-C-RNLP will be satisfied within $(c_{i,s}+1)(L_{max,1}+L_{max,2})$ time units.*

Proof. As in the proof of the original U-C-RNLP bound, there can be at most $c_{i,s}$ rows with requests that conflict with $\mathcal{R}_i$ ahead of $\mathcal{R}_i$. In the worst case, the first such request may have entered *Table* at row $Head+1$ (instead of row *Head*). Therefore, in the worst case, $c_{i,s}+1$ rows of requests must complete before $\mathcal{R}_i$ is satisfied. If $\mathcal{R}_i$ is on Socket 1 (respectively, Socket 2), each request in the row pointed to by *Head* is entitled and becomes satisfied within $L_{max,2}$ (respectively, $L_{max,1}$) time units (Lemma 5.1) and then completes within $L_{max,1}$ (respectively, $L_{max,2}$) times units. Thus, from the time the requests are entitled until they have completed is $(L_{max,1}+L_{max,2})$ time units. Once all requests in that row have completed, *Head* is moved to point to the next row, and all those requests are entitled. Because $c_{i,s}+1$ rows of requests may be entitled and then satisfied before $\mathcal{R}_i$'s row, $\mathcal{R}_i$ will be satisfied within $(c_{i,s}+1)(L_{max,1}+L_{max,2})$ time units in the worst case. □

Reasoning about the G-C-RNLP requires considering the capacity as defined in Chapter 4; the *capacity* of a position is essentially the longest critical section length a request could have to be inserted in that position in the queue without delaying previously issued requests (required by all C-RNLP variants).

Lemma 5.2. *Under the G-C-RNLP, a request $\mathcal{R}_i$ on Socket s becomes satisfied within $L_{max,1}+L_{max,2}$ time units after becoming entitled.*

Proof. In the worst case, Socket s is active and an entitled request $\mathcal{R}_i$ on that socket cannot finish before the end of the phase (based on the phase and dictated by Rule S3). Instead, it must wait for this phase to complete (at most $L_{max,s}$) and for a phase from the other socket to complete (Rules S3 and S4). It total, this is at most $L_{max,1}+L_{max,2}$ time units. □

A group of requests with ordered indices $\mathcal{R}_1$ to $\mathcal{R}_p$ are considered to be *consecutive* if they would be satisfied immediately after one another by following the rules of the chosen C-RNLP variant (that is, for $i \in [1,p]$, $\mathcal{R}_i$ is entitled at the time instance when $\mathcal{R}_{i-1}$ completes).

Lemma 5.3. *A group of requests on Socket s with critical section lengths summing to $\mathcal{L} \leq L_{max,s}$ and being satisfied consecutively will complete within $L_{max,1}+L_{max,2}+\mathcal{L}$ time units.*

Proof. Consider a group of p consecutive requests, indexed in the order in which they are enqueued. Thus, $\mathcal{R}_1$ is the first request to be satisfied. Consider an arbitrary request in the group $\mathcal{R}_i$ that is not the first request.

Suppose that $\mathcal{R}_1$ was satisfied at time t' in the phase that started at time t. The phase in which $\mathcal{R}_i$ was satisfied started at time t''. If all requests ran in the phase starting at time t, they would clearly complete within $L_{max,1} + L_{max,2} + \mathcal{L}$ time units by Lemma 5.2.

Suppose instead $\mathcal{R}_i$ is the first request that cannot execute in the same phase as $\mathcal{R}_1$. There are two potential causes for this. (1) $\mathcal{R}_i$ is not entitled because another request is blocking $\mathcal{R}_i$, and this request delays it beyond when it ought to be satisfied. This situation cannot occur; the requests considered are consecutive, and neither C-RNLP variant would allow a request to be inserted that delayed $\mathcal{R}_i$. (2) $\mathcal{R}_i$ is entitled but would not complete before the end of the phase that $\mathcal{R}_1$ was in that started at time t. Given that there are a series of requests to be satisfied, the phase that started at t will be active until time $t + L_{max}$. The length of time of that requests $\mathcal{R}_1$ through $\mathcal{R}_i$ will execute is $\lambda_i = \sum_{y=1}^{i} L_y$. Thus, if $\mathcal{R}_i$ cannot execute in the same phase as $\mathcal{R}_1$, $t + L_{max,s} < t' + \lambda_i$. Therefore, we can conclude that $\mathcal{R}_1$ did not wait for any time after becoming entitled before becoming satisfied ($\lambda_i - L_{max,s} < 0 \Rightarrow t < t'$, so $\mathcal{R}_i$ was satisfied sometime in the middle of the phase and did not have to wait after becoming entitled to become satisfied). Thus, when $\mathcal{R}_i$ could be satisfied under the C-RNLP variant, but is delayed because of the restrictions on phases, $\mathcal{R}_i$ is entitled and will be satisfied within $L_{max,1} + L_{max,2}$ time units (Lemma 5.2).

Given that $\mathcal{R}_i$ was delayed into a later phase, such a delay (time between becoming entitled and satisfied) cannot occur for any $\mathcal{R}_j$ such that $j > i$. No request can delay it enough to force it into a later phase (as argued in (1)), and it will certainly be able to complete in the same phase as $\mathcal{R}_i$, which started at t''; the phase ends at $t'' + L_{max,s}$, and $t'' + \sum_{y=j}^{p} L_y < t'' + \mathcal{L} < t'' + L_{max,s}$, so this request will complete in this phase.

Since $\mathcal{R}_i$ was the first request to experience delay and no later request experiences delay, a group of consecutive requests on Socket s with critical section lengths summing to $\mathcal{L} \leq L_{max,s}$ will complete within $L_{max,1} + L_{max,2} + \mathcal{L}$ time units. □

Theorem 5.12. *A request $\mathcal{R}_i$ on Socket 1 (respectively, Socket 2) that is serviced by a local lock server running the G-C-RNLP will be satisfied within $c_{i,1}(3L_{max,1} + 2L_{max,2} + L_i)$ (respectively, $c_{i,1}(2L_{max,1} + 3L_{max,2} + L_i)$) time units.*

Proof. The bound on the G-C-RNLP was established in Chapter 4 by considering that a request $\mathcal{R}_i$ may block behind at most c_i conflicting requests with at most c_i positions that would allow satisfaction between those conflicting requests into which $\mathcal{R}_i$ could not be inserted without increasing the blocking. Thus, these positions have a *capacity* less than L_i. (This yielded the original bound of $c_i(L_{max} + L_i$.) Let us now reason about those same two components: the conflicting requests and the positions that are too small.

Consider $\mathcal{R}_i$ on Socket 1 (respectively, Socket 2). The conflicting requests each execute for up to $L_{max,1}$ (respectively, $L_{max,2}$) time units after becoming satisfied. Additionally, each may block for up to $L_{max,1} + L_{max,2}$ time units while entitled before becoming satisfied. In total, these requests can cause $\mathcal{R}_i$ to block for $c_{i,1}(2L_{max,1} + L_{max,2})$ (respectively, $c_{i,1}(L_{max,1} + 2L_{max,2})$) time units.

Next, consider the positions with capacities too small. These positions are created by groups of requests that do not conflict with $\mathcal{R}_i$ but prevent the requests that conflict with $\mathcal{R}_i$ from being satisfied earlier. In the worst case, there is a group of consecutive requests with lengths summing to at most L_i for each such position. Each of these groups contribute up to $c_{i,1}(L_{max,1} + L_{max,2} + L_i)$ time units of blocking.

Thus, in total, the blocking of $\mathcal{R}_i$ is upper bounded by $c_{i,1}(3L_{max,1} + 2L_{max,2} + L_i)$ when $\mathcal{R}_i$ is on Socket 1. Similarly, if $\mathcal{R}_i$ were on Socket 2, its worst-case blocking would be upper bounded by $c_{i,1}(2L_{max,1} + 3L_{max,2} + L_i)$ □

The bounds given in Theorem 5.11 and Theorem 5.12 have implications regarding how to partition a workload under schedulers that assign tasks to execute on specific cores or clusters of cores. This point is illustrated in the context of the U-C-RNLP with the following example.

To begin, suppose that the requests for each resource can be evenly split between sockets such that $L_{max,1} = L_{max,2} = L_{max}$. Then, $c_{i,1} = c_{i,2} = \frac{1}{2}c_i$, and the blocking bound in Theorem 5.11 reduces to $(\frac{1}{2}c_i + 1)(2L_{max}) = (c_i + 2)L_{max}$, which is only one critical-section length longer than that for the original protocol.

While splitting contention evenly like this may be desirable, a system designer could instead choose to assign tasks so as to decrease $c_{i,1}$ at the expense of $c_{i,2}$, which may be a more effective strategy if critical sections of different lengths exist. To see this, suppose that a fraction α of all requests have critical sections of at most $\beta \cdot L_{max}$ time units, where $0 < \beta \leq 1$. If tasks can be assigned so that these shorter requests are all issued from Socket 1 and all others from Socket 2, then the bound from Theorem 5.11 becomes $(\alpha c_i + 1)(\beta L_{max} + L_{max}) = (\alpha c_i + 1)(\beta + 1)L_{max}$ when applied to Socket 1, and $((1-\alpha)c_i + 1)(\beta L_{max} + L_{max}) = ((1-\alpha)c_i + 1)(\beta + 1)L_{max}$ for Socket 2. Depending on the system, such a task assignment could lower the bounds applicable to all requests, as seen in the following example.

Example 5.5. Suppose $c_i = 10$, $L_{max} = 100\mu$s, $\alpha = \frac{1}{5}$, and $\beta = \frac{1}{10}$. With the partitioning of requests described above, the bound on Socket 1 is $(\frac{1}{5} \cdot 10 + 1)(\frac{1}{10} \cdot 100 + 100)\mu s = 330\mu s$, and the bound on Socket 2 is $990\mu s$, both of which are lower than the bound of $(c_i + 1)L_{max} = (10 + 1)100 = 1100\mu s$ for a server-less system (recall the U-C-RNLP discussion in Section 5.1). ◊

Note that the improvement in the above example holds for both sockets, not just the one with lower critical-section lengths.

5.6 Chapter Summary

This chapter considers the use of lock servers on large multicore platforms to lessen overhead associated with contention-sensitive real-time locking protocols. Four specific lock-server paradigms were presented in the context of both the U-C-RNLP and the G-C-RNLP.

Additionally, an experimental study of all four paradigms applied to both C-RNLP variants demonstrated the overhead reductions enabled by these paradigms. In these experiments, the use of lock servers often reduced overhead dramatically. For example, the paradigm that generally performed best, static global lock servers, typically exhibited overhead reductions in the range of 25%–75% compared to not using lock servers.

Finally, this chapter presented the use of a reader/reader locking protocol to coordinate between multiple local lock servers, along with the associated blocking analysis and a discussion of the related tradeoffs.

Acknowledgment. The work presented in this chapter originally appeared in a paper written by Tanya Amert and myself. In addition to being involved throughout the project, Tanya implemented the lock servers, conducted the evaluation, and analyzed the resulting data.

CHAPTER 6: CONCLUSION

Nested resource requests enable a task to access multiple resources concurrently. Unfortunately, most existing approaches that grant nested resource access result in high blocking, which can cause tasks to miss deadlines. The primary challenge in supporting nested resource access is the Transitive Blocking Chain Problem discussed in Section 1.3. Requests can enqueue in a manner that forms a long chain of blocking, even when many requests do not require an overlapping set of resources. The blocking chains formed by nested requests also impact read requests and non-nested write requests.

Transitive blocking chains can be broken by reordering requests, but this can cause significant protocol overhead. Both overhead and blocking must be considered in schedulability analysis. These observations motivated the development of locking protocols efficient in both overhead and blocking, with a focus on contention-sensitive protocols.

The remainder of this chapter summarizes the results of this dissertation (Section 6.1), other related work of the author that was not included in this dissertation (Section 6.2), and directions for future work (Section 6.3).

6.1 Summary of Results

This section summarizes the key contributions of this dissertation.

Efficient resource access for read requests and non-nested write requests. In systems with nested requests, both read requests and non-nested write requests can experience significant blocking. However, it is relatively straightforward to grant contention-sensitive resource access in the absence of nested resource access. This observation motivated the development of the fast RW-RNLP, which was presented in Chapter 3. The fast RW-RNLP provides a fast-path mechanism for read requests and non-nested write requests. By handling each type of request separately, it prevents transitive blocking chains from impacting read and non-nested write requests.

This separation of requests requires the use of an arbitration mechanism between request types. Two arbitration mechanisms were presented: the R^3LP and the RW-RNLP*. The protocol as a whole was evaluated on the basis of schedulability. In the study presented in Chapter 3, the fast RW-RNLP resulted in higher schedulability than prior protocols when non-nested resource access was the common case. Additionally, the study showed that the fast RW-RNLP with the R^3LP arbitration mechanism tended to outperform the fast RW-RNLP with the RW-RNLP*.

Efficient resource access for nested write requests. Chapter 4 presented two protocols to handle nested write requests. Both protocols reorder requests as they are issued in order to reduce blocking. The C-RNLP is the first multiprocessor real-time locking protocol to grant contention-sensitive nested resource access. A newly issued request is inserted into the earliest queue position that ensures safety and preserves delays for previously issued requests. This cutting-ahead mechanism breaks transitive blocking chains. Two variants of the C-RNLP were presented. The general variant, the G-C-RNLP, requires maintaining a node in the appropriate resource queue(s) for each active request and considers the different per-request maximum critical-section lengths when determining when a newly issued request will be satisfied. The uniform variant, the U-C-RNLP, instead maintains a set of slots in which requests may be satisfied, each of which allows one request to be satisfied per resource and for a duration of up to the maximum critical-section length. This simplifies some of the protocol logic, which in turn results in slightly lower overhead than the general variant. These two C-RNLP variants both result in contention-sensitive blocking, with slightly different blocking bounds. In the presented schedulability study, both variants outperformed existing approaches, and the G-C-RNLP tended to result in higher schedulability than the U-C-RNLP.

Chapter 4 also presented the CGLP, which was developed in response to the higher protocol overhead of C-RNLP and the description of an additional challenge with granting nested write access: the Request Timing Problem. The CGLP is comprised of an offline component that determines the concurrency groups for a given task system and an online component that arbitrates between these groups. Chapter 4 presented several options for the offline component that optimize the group creation based on task system parameters in order to reduce the blocking. The online component coordinates request satisfaction by group with low overhead. The CGLP outperformed existing approaches in the presented schedulability study, and in many scenarios, outperformed the C-RNLP variants. In the scenarios in which the CGLP did not result in the highest schedulability, the G-C-RNLP resulted in the highest schedulability.

Protocol-independent overhead reduction. Lock servers were presented in Chapter 5 as a protocol-independent method of reducing protocol overhead. Locking protocols must store some state in order to safely grant resources access. The accessing of protocol-specific data structures causes a portion of the overall protocol overhead; when this state is not cached, it must be loaded from memory. While cache evictions can be caused by unrelated task execution, the execution of the locking protocol itself invalidates cached data on other cores when updating the lock state.

Lock servers are dedicated processes that execute locking protocol logic on behalf of tasks in order to reduce cache evictions and thereby reduce overhead of the protocol. Chapter 5 presented four configurations for lock servers, distinguished by their locality and mobility. These configurations are based on knowledge of the underlying cache structure and offer different tradeoffs. The development of lock severs was motivated by the high overhead of the C-RNLP variants, which is caused in part by the additional timing information about requests that must be maintained. When lock servers were applied to the C-RNLP variants, they significantly reduced overhead, especially under the configuration with a static global lock server.

6.2 Other Related Work

Two publications to which the author contributed fall beyond the scope of this dissertation. These contributions are summarized briefly here.

Replicated resources.[1] The use of a k-exclusion locking protocol enables granting lock access to k replicas of a resource, granting access to one replica per request. This sharing paradigm can be extended to a more general notion of *replicated resources*, in which there are k replicas and a task may request between one and k of these replicas. This work was motivated by two use cases for which the primary source of delays is caused by different factors. One use case has long critical-section lengths, and blocking is the limiting factor in performance. For the other use case, overhead has a more significant impact on schedulability.

To address these challenges, we presented three replica allocation protocols. Two of these achieve low overhead at the expense of non-optimal blocking. The third results in optimal blocking (ignoring constant factors), but at the cost of higher overhead. The low blocking approach uses the notion of cutting-ahead, but

[1] This work previously appeared in:

Nemitz, C., Yang, K., Yang, M., Ekberg, P., and Anderson, J. (2016). Multiprocessor real-time locking protocols for replicated resources. In *Proceedings of the 28th Euromicro Conference on Real-Time Systems*.

for replicas instead of nested resources. Additionally, we distinguished between the allocation and assignment problems: a task requiring replica *allocation* simply needs to be granted arbitrary replicas, while a task requiring replica *assignment* must know which replica(s) it was granted. For example, tokens that limit system contention may be allocated, but assignment is required for granting access to specific GPUs or other accelerators.

Phase-fair locking.[2] The phase-fair reader/writer locking protocol coordinates non-nested read and write access to a shared resource (Brandenburg and Anderson, 2010). It differs from prior reader/write protocols in the way unsatisfied read and write requests are ordered. The *phase-fair* approach alternates between read phases, in which all active read requests are satisfied, and write phases, in which a single active write request is satisfied. This results in better blocking bounds for real-time systems than prior approaches (some of which have unbounded blocking).

Using an existing implementation of the phase-fair locking protocol revealed that throughput in a read-dominated or a read-only workload was much lower than expected. Further investigation showed that read requests interfere in updating the protocol state, resulting in overhead that reduces throughput. Each newly issued read request updates a global count of read requests, frequently requiring a remote memory reference and invalidating the cached protocol state on other cores. This insight led us to develop a new implementation of the phase-fair locking protocol. Our implementation uses per-core variables for maintaining protocol state on active read requests, ensuring isolation between read requests, which dramatically reduced overhead. This requires write requests to check additional protocol data structures, resulting in higher overhead for write requests. This tradeoff resulted in manageable overhead and significantly improved throughput.

Use of the phase-fair locking protocol in a real-time system also necessitates incorporating its blocking analysis into schedulability analysis. Existing blocking analysis for phase-fair locking was inflation-based, inflating each task by the blocking it may incur. We instead provided inflation-free blocking analysis that extends prior analysis (Biondi and Brandenburg, 2016). This analysis fits into an existing schedulability framework, which requires the maximum synchronization delay during a given interval. Thus, we produced an optimization problem that maximizes these delays subject to constraints on how read requests and

[2]This work previously appeared in:

Nemitz, C., Caspin, S., Anderson, J., and Ward, B. (2021b). Light reading: Optimizing reader/writer locking for read-dominant real-time workloads. In *Proceedings of the 33rd Euromicro Conference on Real-Time Systems*.

write requests can interact under the phase-fair locking protocol. This new inflation-free approach yielded significant improvements in a schedulability study.

6.3 Future Work

There are multiple directions for future work based on the contributions of this dissertation.

Inflation-free blocking analysis. As mentioned briefly above, recent work has incorporated inflation-free blocking analysis into schedulability analysis for a few protocols. An extension to this work would be to specify optimization problems that correspond to the protocols in this dissertation. As these protocols reorder requests, new constraints must be developed to reflect the request orderings. For example, for the C-RNLP variants, requests cannot be assumed to execute in FIFO order, but constraints could be developed to reflect that safety and delay preservation are always upheld.

Blocking and overhead comparison. With the inflation-based analysis used in this dissertation, the impact of overhead was magnified by the computed blocking when checking schedulability. A future study could separate these delays in order to determine which has the greater impact on certain types of scenarios. This knowledge could be especially helpful for task systems that are not deemed schedulable. For example, it could indicate that the blocking is too high, regardless of the overhead, or that the overhead is too high relative to the critical-section lengths.

Suspension-based protocol variants. For some applications, suspension-based protocols are likely to result in better schedulability than spin-based ones. For example, if critical sections are long relative to some task periods, it may be beneficial for a blocked task to yield the processor to another task. Suspension-based protocols require different progress mechanisms and blocking analysis.

To reap the full benefit of a suspension-based approach, several aspects must be explored. For example, it may be possible to separate requests by duration, with some requests spinning and others suspending (Ward and Anderson, 2013).

Under the fast RW-RNLP, blocking can occur in two stages: first in the type-specific component, and then in the global arbitration mechanism. Thus, developing a suspension-based version of the fast RW-RNLP requires implementing a progress mechanism to reflect this. This two-staged blocking also gives two decision points for spinning or suspending.

For the C-RNLP, the decision for certain requests to spin or suspend could be made at runtime, as the protocol already maintains timing information on all active requests. This would necessitate blocking analysis that incorporate these options in addition to the potential for requests to cut ahead.

Lock servers were developed for spin-based locking protocols. However, the notion of a lock server lends itself to a kernel-based implementation. A dedicated task could be established to act as a lock server, and this task may be able to run alongside other kernel services. Transitioning to suspension-based protocols would require adjusting some assumptions. For example, in the spin-based version, a blocked (spinning) task can temporarily serve as the lock server. Enabling this behavior for a suspension-based protocol would require updating the schedulability analysis to reflect that all but one blocked task will suspend. Additionally, there may exist alternate lock server configurations that better reflect the needs of a suspension-based protocol.

Evaluation with other scheduling algorithms. Ultimately, in hard real-time systems, locking protocols are evaluated on the basis of enabling schedulability for task systems. Schedulability is impacted by system design decisions beyond the locking protocol and its resulting overhead and blocking. These decisions include the chosen scheduler along with any partitioning or cluster assignments.

The schedulability studies presented in this dissertation assumed the use of G-EDF. However, other scheduling algorithms introduce additional questions. For example, can the blocking analysis of each of the presented protocols be refined for a clustered system?

Considering the use of lock servers under different scheduling algorithms also raises new possibilities. For instance, some lock server configurations require a dedicated task pinned to a single core. This type of tradeoff (losing a core for general processing to reduce protocol overhead) can be investigated in the context of different scheduling choices. To yield a benefit, it might be necessary to *provably* ensure that lock servers always execute in cache. Such assurances could be provided by integrating lock servers with cache-isolation techniques explored elsewhere (Altmeyer et al., 2014; Campoy et al., 2001; Chisholm et al., 2015; Herter et al., 2011; Kim et al., 2013; Kirk and Strosnider, 1990; Ward et al., 2013; Xu et al., 2016, 2017; Yun et al., 2014). Finally, the amount of cache space to dedicate to a lock server may depend on the scheduling algorithm selection and the tradeoff of instead allowing tasks to use that space.

BIBLIOGRAPHY

Afshar, S., Behnam, M., Bril, R., and Nolte, T. (2014). Flexible spin-lock model for resource sharing in multiprocessor real-time systems. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems*.

Afshar, S., Behnam, M., Bril, R., and Nolte, T. (2017). An optimal spin-lock priority assignment algorithm for real-time multi-core systems. In *Proceedings of the 23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*.

Afshar, S., Behnam, M., Bril, R., and Nolte, T. (2018). Per processor spin-based protocols for multiprocessor real-time systems. *Leibniz Transactions on Embedded Systems*, 4(2):03:1–03:30.

Afshar, S., Behnam, M., and Nolte, T. (2013). Integrating independently developed real-time applications on a shared multi-core architecture. *ACM SIGBED Review*, 10(3):49–56.

Afshar, S., Khalilzad, N., Nemati, F., and Nolte, T. (2015). Resource sharing among prioritized real-time applications on multiprocessors. *ACM SIGBED Review*, 12(1):46–55.

Altmeyer, S., Douma, R., Lunniss, W., and Davis, R. (2014). Evaluation of cache partitioning for hard real-time systems. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems*.

Anderson, J., Bud, V., and Devi, U. (2005). An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*.

Anderson, J., Jain, R., and Ramamurthy, S. (1997). Wait-free object-sharing schemes for real-time uniprocessors and multiprocessors. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*.

Anderson, J. and Ramamurthy, S. (1996). A framework for implementing objects and scheduling tasks in lock-free real-time systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*.

Attiya, H., Bar-Noy, A., Dolev, D., Koller, D., Peleg, D., and Reischuk, R. (1987). Achievable cases in an asynchronous environment. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*.

AUTOSAR (2019). AUTOSAR Release 4.4, Classic Platform, Specification of Operating System. `https://www.autosar.org/`.

Bacon, D., Konuru, R., Murthy, C., and Serrano, M. (1998). Thin locks: Featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*.

Baker, T. (1991). Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99.

Baker, T. (2003). Multiprocessor EDF and deadline monotonic schedulability analysis. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*.

Baker, T. (2005). An analysis of EDF schedulability on a multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 16(8):760–768.

Baker, T. and Shaw, A. (1988). The cyclic executive model and Ada. In *Proceedings of the 9th IEEE Real-Time Systems Symposium*.

Baker, T. and Shaw, A. (1989). The cyclic executive model and Ada. *Real-Time Systems*, 1(1):7–25.

Bandh, T., Carle, G., and Sanneck, H. (2009). Graph coloring based physical-cell-ID assignment for LTE networks. In *Proceedings of the 2009 International Wireless Communications and Mobile Computing Conference*.

Barnier, N. and Brisset, P. (2004). Graph coloring for air traffic flow management. *Annals of Operations Research*, 130(1-4):163–178.

Barros, A., Pinho, L., and Yomsi, P. (2015). Non-preemptive and SRP-based fully-preemptive scheduling of real-time software transactional memory. *Journal of Systems Architecture*, 61(10):553–566.

Baruah, S. (2007). Techniques for multiprocessor global schedulability analysis. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*.

Baruah, S. and Brandenburg, B. (2013). Multiprocessor feasibility analysis of recurrent task systems with specified processor affinities. In *Proceedings of the 34th IEEE Real-Time Systems Symposium*.

Bastoni, A., Brandenburg, B., and Anderson, J. (2011). Is semi-partitioned scheduling practical? In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*.

Belwal, C. and Cheng, A. (2011). Lazy versus eager conflict detection in software transactional memory: A real-time schedulability perspective. *Embedded Systems Letters*, 3(1):37–41.

Bertogna, M. and Baruah, S. (2011). Tests for global EDF schedulability analysis. *Journal of Systems Architecture*, 57(5):487–497.

Bertogna, M., Cirinei, M., and Lipari, G. (2005). Improved schedulability analysis of EDF on multiprocessor platforms. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*.

Biondi, A. and Brandenburg, B. (2016). Lightweight real-time synchronization under P-EDF on symmetric and asymmetric multiprocessors. In *Proceedings of the 28th Euromicro Conference on Real-Time Systems*.

Biondi, A., Brandenburg, B., and Wieder, A. (2016). A blocking bound for nested FIFO spin locks. In *Proceedings of the 37th IEEE Real-Time Systems Symposium*.

Block, A., Leontyev, H., Brandenburg, B., and Anderson, J. (2007). A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*.

Bonifaci, V., Brandenburg, B., D'Angelo, G., and Marchetti-Spaccamela, A. (2016). Multiprocessor real-time scheduling with hierarchical processor affinities. In *Proceedings of the 28th Euromicro Conference on Real-Time Systems*.

Bonifaci, V., D'Angelo, G., and Marchetti-Spaccamela, A. (2017). Algorithms for hierarchical and semi-partitioned parallel scheduling. In *Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium*.

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

Brandenburg, B. (2014). The FMLP+: An asymptotically optimal real-time locking protocol for suspension-aware analysis. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems*.

Brandenburg, B. (2019). Multiprocessor real-time locking protocols: A systematic review. *arXiv preprint arXiv:1909.09600*.

Brandenburg, B. and Anderson, J. (2007). Feather-trace: A lightweight event tracing toolkit. In *Proceedings of the 3rd International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*.

Brandenburg, B. and Anderson, J. (2008a). A comparison of the M-PCP, D-PCP, and FMLP on LITMUS$^{\mathrm{RT}}$. In *Proceedings of the International Conference on Principles of Distributed Systems*.

Brandenburg, B. and Anderson, J. (2008b). An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS$^{\mathrm{RT}}$. In *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*.

Brandenburg, B. and Anderson, J. (2010). Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-Time Systems*, 46(1):25–87.

Brandenburg, B. and Anderson, J. (2011). Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In *Proceedings of the 9th ACM International Conference on Embedded Software*.

Brandenburg, B. and Anderson, J. (2013). The OMLP family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Systems*, 17(2):277–342.

Brandenburg, B., Calandrino, J., Block, A., Leontyev, H., and Anderson, J. (2008). Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*.

Burns, A. and Wellings, A. (2013). A schedulability compatible multiprocessor resource sharing protocol - MrsP. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*.

Campoy, M., Ivars, A., and Busquets-Mataix, J. (2001). Static use of locking caches in multitask preemptive real-time systems. In *Proceedings of the IEEE/IEE Real-Time Embedded Systems Workshop*.

Cerqueira, F., Gujarati, A., and Brandenburg, B. (2014). Linux's processor affinity API, refined: Shifting real-time tasks towards higher schedulability. In *Proceedings of the 35th IEEE Real-Time Systems Symposium*.

Chaitin, G., Auslander, M., Chandra, A., Cocke, J., Hopkins, M., and Markstein, P. (1981). Register allocation via coloring. *Computer Languages*, 6(1):47–57.

Chang, Y., Davis, R., and Wellings, A. (2010). Reducing queue lock pessimism in multiprocessor schedulability analysis. In *Proceedings of the 18th International Conference on Real-Time Networks and Systems*.

Chen, C. and Tripathi, S. (1994). Multiprocessor priority ceiling based protocols. Department of Computer Science, University of Maryland. Technical report, CS-TR-3252.

Chisholm, M., Ward, B., Kim, N., and Anderson, J. (2015). Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *Proceedings of the 36th IEEE Real-Time Systems Symposium*.

Cho, H., Ravindran, B., and Jensen, E. D. (2007). Space-optimal, wait-free real-time synchronization. *IEEE Transactions on Computers*, 56(3):373–384.

Courtois, P., Heymans, F., and Parnas, D. (1971). Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667–668.

Dijkstra, E. (1978). Two starvation free solutions to a general exclusion problem. EWD 625, Plataanstraat 5, 5671 Al Nuenen, The Netherlands.

El-Shambakey, M. (2013). *Real-Time Software Transactional Memory: Contention Managers, Time Bounds, and Implementations*. PhD thesis, Virginia Polytechnic Institute.

Elliott, G. and Anderson, J. (2013). An optimal *k*-exclusion real-time locking protocol motivated by multi-GPU systems. *Real-Time Systems*, 49(2):140–170.

Faggioli, D., Lipari, G., and Cucinotta, T. (2010). The multiprocessor bandwidth inheritance protocol. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*.

Faggioli, D., Lipari, G., and Cucinotta, T. (2012). Analysis and implementation of the multiprocessor bandwidth inheritance protocol. *Real-Time Systems*, 48(6):789–825.

Gai, P., Di Natale, M., Lipari, G., Ferrari, A., Gabellini, C., and Marceca, P. (2003). A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*.

Gai, P., Lipari, G., and Di Natale, M. (2001). Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*.

Garrido, J., Zhao, S., Burns, A., and Wellings, A. (2017). Supporting nested resources in MrsP. In *Proceedings of the Ada-Europe International Conference on Reliable Software Technologies*.

Goossens, J., Funk, S., and Baruah, S. (2003). Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2):187–205.

Gurobi (2018). Gurobi optimizer reference manual. `http://www.gurobi.com`.

Havender, J. (1968). Avoiding deadlock in multitasking systems. *IBM Systems Journal*, 7(2):74–84.

Herlihy, M. (1991). Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149.

Herlihy, M. and Wing, J. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492.

Herter, J., Backes, P., Haupenthal, F., and Reineke, J. (2011). CAMA: A predictable cache-aware memory allocator. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*.

Hsiu, P., Lee, D., and Kuo, T. (2011). Task synchronization and allocation for many-core real-time systems. In *Proceedings of the 9th ACM International Conference on Embedded Software*.

Huang, W., Yang, M., and Chen, J. (2016). Resource-oriented partitioned scheduling in multiprocessor systems: How to partition and how to share? In *Proceedings of the 37th IEEE Real-Time Systems Symposium*.

Jarrett, C., Ward, B., and Anderson, J. (2015). A contention-sensitive fine-grained locking protocol for multiprocessor real-time systems. In *Proceedings of the 23rd International Conference on Real-Time Networks and Systems*.

Joung, Y. (2000). Asynchronous group mutual exclusion. *Distributed Computing*, 13(4):189–206.

Keane, P. and Moir, M. (1999). A simple local-spin group mutual exclusion algorithm. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*.

Keane, P. and Moir, M. (2001). A simple local-spin group mutual exclusion algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 12(7):673–685.

Kim, H., Kandhalu, A., and Rajkumar, R. (2013). A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*.

Kirk, D. and Strosnider, J. (1990). SMART (strategic memory allocation for real-time) cache design using the MIPS R3000. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*.

Lakshmanan, K., de Niz, D., and Rajkumar, R. (2009). Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*.

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

Lipari, G., Lamastra, G., and Abeni, L. (2004). Task synchronization in reservation-based real-time systems. *IEEE Transactions on Computers*, 53(12):1591–1601.

Liu, C. and Layland, J. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61.

Liu, J. (2000). *Real-Time Systems*. Prentice Hall.

Lozi, J., David, F., Thomas, G., Lawall, J., and Muller, G. (2012). Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *Proceedings of the 2012 USENIX Annual Technical Conference*.

Marx, D. (2004). Graph colouring problems and their applications in scheduling. *Periodica Polytechnica Electrical Engineering*, 48(1-2):11–16.

Mellor-Crummey, J. and Scott, M. (1991a). Algorithms for scalable synchronization of shared-memory multiprocessors. *Transactions on Computer Systems*, 9(1):21–65.

Mellor-Crummey, J. and Scott, M. (1991b). Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the 3rd ACM Symposium on Principles and Practice of Parallel Programming*.

Moir, M. and Anderson, J. (1995). Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39.

Mok, A. (1983). *Fundamental design problems of distributed systems for the hard-real-time environment*. PhD thesis, Massachusetts Institute of Technology.

Nemati, F., Behnam, M., and Nolte, T. (2011). Independently-developed real-time systems on multi-cores with shared resources. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*.

Nemitz, C., Amert, T., and Anderson, J. (2017). Real-time multiprocessor locks with nesting: Optimizing the common case. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*.

Nemitz, C., Amert, T., and Anderson, J. (2018). Using lock servers to scale real-time locking protocols: Chasing ever-increasing core counts. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems*.

Nemitz, C., Amert, T., and Anderson, J. (2019a). Real-time multiprocessor locks with nesting: Optimizing the common case. *Real-Time Systems*, 55(2):296–348.

Nemitz, C., Amert, T., Goyal, M., and Anderson, J. (2019b). Concurrency groups: A new way to look at real-time multiprocessor lock nesting. In *Proceedings of the 27th International Conference on Real-Time Networks and Systems*.

Nemitz, C., Amert, T., Goyal, M., and Anderson, J. (2021a). Concurrency groups: A new way to look at real-time multiprocessor lock nesting. *Real-Time Systems*, 57(1):190–226.

Nemitz, C., Caspin, S., Anderson, J., and Ward, B. (2021b). Light reading: Optimizing reader/writer locking for read-dominant real-time workloads. In *Proceedings of the 33rd Euromicro Conference on Real-Time Systems*.

Nemitz, C., Yang, K., Yang, M., Ekberg, P., and Anderson, J. (2016). Multiprocessor real-time locking protocols for replicated resources. In *Proceedings of the 28th Euromicro Conference on Real-Time Systems*.

Palladino, S. (2010). Modelling graph coloring with integer linear programming. `https://manas.tech/blog/2010/09/16/modelling-graph-coloring-with-integer-linear-programming.html`.

Rajkumar, R. (1990). Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings of the 10th International Conference on Distributed Computing Systems*.

Rajkumar, R. (1991). Synchronization in real-time systems: A priority inheritance approach. Kluwer Academic Publishers.

Rajkumar, R., Sha, L., and Lehoczky, J. (1988). Real-time synchronization protocols for multiprocessors. In *Proceedings of the 9th IEEE Real-Time Systems Symposium*.

Ramamurthy, S. (1997). *A Lock-Free Approach to Object Sharing in Real-Time Systems*. PhD thesis, University of North Carolina at Chapel Hill.

Rhee, I. and Martin, G. R. (1995). A scalable real-time synchronization protocol for distributed systems. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*.

Sarni, T., Queudet, A., and Valduriez, P. (2009). Real-time support for software transactional memory. In *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*.

SchedCAT (2019). SchedCAT: Schedulability test collection and toolkit. `https://github.com/brandenburg/schedcat`.

Schoeberl, M., Brander, F., and Vitek, J. (2010). RTTM: Real-time transactional memory. In *Proceedings of the 25th ACM Symposium on Applied Computing*.

Schoeberl, M. and Hilber, P. (2010). Design and implementation of real-time transactional memory. In *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*.

Schwan, K. and Zhou, H. (1992). Multiprocessor real-time threads. *ACM SIGOPS Operating Systems Review*, 26(1):54–65.

Sha, L., Rajkumar, R., and Lehoczky, J. P. (1990). Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers*, 39(9):1175–1185.

Takada, H. and Sakamura, K. (1995). Real-time scalability of nested spin locks. In *Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications*.

Tang, S. and Anderson, J. (2020). Towards practical multiprocessor EDF with affinities. In *Proceedings of the 41st IEEE Real-Time Systems Symposium*.

Voronov, S. and Anderson, J. (2018). An optimal semi-partitioned scheduler assuming arbitrary affinity masks. In *Proceedings of the 39th IEEE Real-Time Systems Symposium*.

Wang, C., Takada, H., and Sakamura, K. (1996). Priority inheritance spin locks for multiprocessor real-time systems. In *Proceedings of the 2nd International Symposium on Parallel Architectures, Algorithms, and Networks*.

Ward, B. (2015). Relaxing resource-sharing constraints for improved hardware management and schedulability. In *Proceedings of the 36th IEEE Real-Time Systems Symposium*.

Ward, B. (2016). *Sharing Non-Processor Resources in Multiprocessor Real-Time Systems*. PhD thesis, University of North Carolina at Chapel Hill.

Ward, B. and Anderson, J. (2012). Supporting nested locking in multiprocessor real-time systems. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*.

Ward, B. and Anderson, J. (2013). Fine-grained multiprocessor real-time locking with improved blocking. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*.

Ward, B. and Anderson, J. (2014). Multi-resource real-time reader/writer locks for multiprocessors. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*.

Ward, B., Elliott, G., and Anderson, J. (2012). Replica-request priority donation: A real-time progress mechanism for global locking protocols. In *Proceedings of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*.

Ward, B., Herman, J., Kenna, C., and Anderson, J. (2013). Making shared caches more predictable on multicore platforms. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*.

Wieder, A. and Brandenburg, B. (2013). On spin locks in AUTOSAR: Blocking analysis of FIFO, unordered, and priority-ordered spin locks. In *Proceedings of the 34th IEEE Real-Time Systems Symposium*.

Wieder, A. and Brandenburg, B. (2014). On the complexity of worst-case blocking analysis of nested critical sections. In *Proceedings of the 35th IEEE Real-Time Systems Symposium*.

Xu, M., Phan, L. T. X., Choi, H.-Y., and Lee, I. (2016). Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation. In *Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium*.

Xu, M., Phan, L. T. X., Choi, H.-Y., and Lee, I. (2017). vCAT: Dynamic cache management using CAT virtualization. In *Proceedings of the 23rd IEEE Real-Time and Embedded Technology and Applications Symposium*.

Yang, M., Wieder, A., and Brandenburg, B. (2015). Global real-time semaphore protocols: A survey, unified analysis, and comparison. In *Proceedings of the 36th IEEE Real-Time Systems Symposium*.

Yoo, R. and Lee, H.-H. (2008). Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*.

Yun, H., Mancuso, R., Wu, Z., and Pellizzoni, R. (2014). PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium*.

Zhao, S., Garrido, J., Burns, A., and Wellings, A. (2017). New schedulability analysis for MrsP. In *Proceedings of the 23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*.

Zhao, S., Garrido, J., Wei, R., Burns, A., Wellings, A., and Juan, A. (2020). A complete run-time overhead-aware schedulability analysis for MrsP under nested resources. *Journal of Systems and Software*, 159:110449.