Real-Time Multiprocessor Locks with Nesting: Optimizing the Common Case

Catherine E. Nemitz, Tanya Amert, and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill

ABSTRACT

In prior work on multiprocessor real-time locking protocols, only protocols within the RNLP family support unrestricted lock nesting while guaranteeing asymptotically optimal priority-inversion blocking bounds. However, these protocols support nesting at the expense of increasing the cost of processing non-nested lock requests, which tend to be the common case in practice. To remedy this situation, a new *fast-path mechanism* is presented herein that extends prior RNLP variants by ensuring that non-nested requests are processed efficiently. This mechanism yields overhead and blocking costs for such requests that are nearly identical to those seen in the most efficient single-resource locking protocols. In experiments, the proposed fast-path mechanism enabled observed blocking times for non-nested requests that were up to 17 times lower than under an existing RNLP variant.

CCS CONCEPTS

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

KEYWORDS

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

ACM Reference format:
Catherine E. Nemitz, Tanya Amert, and James H. Anderson. 2017. Real-Time Multiprocessor Locks with Nesting: Optimizing the Common Case. In *Proceedings of RTNS '17, Grenoble, France, October 4–6, 2017,* 10 pages.
https://doi.org/10.1145/3139258.3139262

1 INTRODUCTION

Multicore technologies have the potential to enable a wealth of new computationally intensive embedded real-time applications, provided efficient resource-allocation infrastructure is available. Such infrastructure must necessarily include support for multiprocessor real-time locking protocols. Evidence suggests that the ability to nest lock requests to allow a task to access multiple resources simultaneously is commonly required in practice, even though non-nested requests predominate [1, 3]. However, only a few protocols exist that support unrestricted nesting, and of those that do, only those in the RNLP (real-time nested locking protocol) family provide asymptotically optimal priority-inversion blocking (pi-blocking) bounds.

The RNLP family includes the basic RNLP [13], which provides mutex sharing, the RW-RNLP [12], which provides reader/writer sharing, and the C-RNLP [9], which provides contention-sensitive mutex sharing. A locking protocol is *contention-sensitive* if a task's pi-blocking time is $O(C)$, where C is the number of tasks actually contending for the same resources [9]. The key to ensuring contention-sensitivity is to avoid *transitive blocking chains*, which are caused by nested requests and may create blocking relationships between otherwise non-conflicting tasks.

To support nested requests, each RNLP variant employs logic more complicated than that of single-resource protocols. This logic is the most complex in the C-RNLP because it ensures contention-sensitivity. The RNLP and the RW-RNLP employ simpler logic but sacrifice contention-sensitive pi-blocking, even for non-nested requests. Thus, these protocols support nesting (the *less common* case) at the expense of increased processing costs and/or pi-blocking bounds for non-nested requests (the *more common* case).

Contributions. Motivated by this observation, we propose a new *fast-path mechanism* for the RNLP family that was designed with the twin goals of ensuring non-nested lock requests are **(i)** contention-sensitive and **(ii)** incur low lock/unlock overheads comparable to those of single-resource protocols. We present this fast-path mechanism in the context of a new reader/writer RNLP variant, which we call the *fast* RW-RNLP.[1] In reader/writer sharing, read requests can execute concurrently but write requests require exclusive access [6]. Since reader/writer sharing subsumes mutex sharing, the fast RW-RNLP can be applied to support the latter.

We build directly on two prior protocols. The first is the *phase-fair ticket lock (PF-TL)*, which is used to provide reader/writer access to a single resource [4]. The PF-TL is a non-preemptive spin-lock. The protected resource has two FIFO request queues, one for reads and one for writes. If both kinds of requests are queued concurrently, the protocol alternates between *read phases* wherein read requests are given preference, and corresponding *write phases.* The PF-TL has asymptotically optimal pi-blocking bounds and very low runtime overheads (and is trivially contention-sensitive).

The other protocol we build on is the RW-RNLP. At this point, it suffices to know that the RW-RNLP uses two queues per resource, one for readers and one for writers, like the PF-TL does for a single resource. However, additional complications arise because tasks can hold multiple resources at the same time. This affects the queueing logic and the orchestration of phases. The latter becomes more difficult because different resources may be in different phases.

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

Work supported by NSF grants CNS 1409175, CPS 1446631, and CNS 1563845, AFOSR grant FA9550-14-1-0161, ARO grant W911NF-14-1-0499, and funding from General Motors. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGS-1650116. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

RTNS '17, October 4–6, 2017, Grenoble, France

ACM ISBN 978-1-4503-5286-4/17/10…$15.00
https://doi.org/10.1145/3139258.3139262

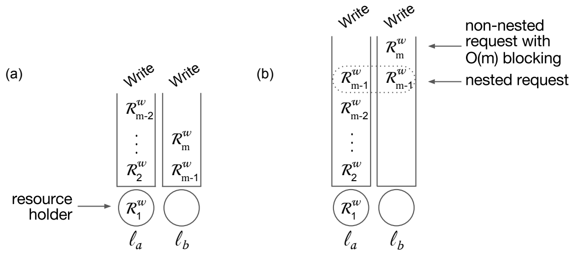


Figure 1: Impact of transitive blocking on non-nested requests. In (b), $\mathcal{R}_{m-1}$ requests ℓ_a and ℓ_b together using a DGL, as allowed by the RW-RNLP.

The more complicated queueing logic of the RW-RNLP causes even non-nested requests to incur higher lock/unlock overheads. Additionally, such requests do not have contention-sensitive pi-blocking bounds because they may become part of transitive blocking chains caused by nested requests. A simple example is given in Fig. 1, which depicts two resources ℓ_a and ℓ_b, on m processors, accessed by m write requests, $\mathcal{R}_1, \ldots, \mathcal{R}_m$, issued in this order. Two scenarios are shown that result in different pi-blocking times for request $\mathcal{R}_m$. In inset (a), there are no nested requests, and each resource is protected by a PF-TL. Here, $\mathcal{R}_m$ is pi-blocked by only one other request, which is clearly in accordance with the definition of contention-sensitivity. In inset (b), request $\mathcal{R}_{m-1}$ accesses both resources, and the RW-RNLP is used. Here, the nested request $\mathcal{R}_{m-1}$ forces $\mathcal{R}_m$ to be pi-blocked by all other requests, which is clearly not contention-sensitive.

In our fast RW-RNLP, non-nested requests are immune from the effects for transitive blocking chains caused by nesting. This is achieved by employing a modular design that mostly separates concerns related to handling nested and non-nested requests. This modular design also facilitates applying the protocol in different contexts. For example, one of the components we introduce directly supports constant-time access for all requests in systems of single-writer, multiple-reader resources, a common use case in embedded systems [8]. Also, by altering one of the components, contention-sensitivity can also be ensured for nested requests (like with the C-RNLP, but at the expense of greater overheads for such requests). Waiting in the fast RNLP can be realized by either spinning or suspension, though we consider only the former in detail due to space constraints. When no nested requests occur, the fast RW-RNLP functions nearly identically to a set of per-resource PF-TLs.

This similarity is borne out in experiments we conducted in which lock/unlock overheads and observed pi-blocking times were recorded for non-nested requests. We found that lock/unlock overheads for such requests were nearly identical under the fast RW-RNLP and PF-TLs. We also found that observed pi-blocking times for such requests were reduced compared to the RW-RNLP. This is because such requests require less overhead and are immune to transitive blocking effects under the fast RW-RNLP.

Organization. In the rest of the paper, we give needed background (Sec. 2), describe the fast RW-RNLP in detail (Sec. 3), discuss our experiments (Sec. 4), and conclude (Sec. 5).

2 BACKGROUND

In this section, we present relevant background material.

Task model. We consider the classic sporadic real-time task model (we assume familiarity with this model) and focus on a system $\Gamma = \{\tau_1, \ldots, \tau_n\}$ of n tasks scheduled on m processors by a job-level fixed-priority scheduler (*e.g.* partitioned, global, or clustered earliest-deadline-first). We denote an arbitrary job of task τ_i as J_i.

Resource model. We assume the existence of n_r shared resources, denoted $\mathcal{L} = \{\ell_1, \ldots, \ell_{n_r}\}$. When a job J_i requires access to one or more of these resources, it *issues* a *request* $\mathcal{R}_i$ for its needed resources by invoking a locking protocol. We say that $\mathcal{R}_i$ is *satisfied* as soon as J_i *holds* its requested resources and that it has *completed* once J_i has *released* all of those resources. A request $\mathcal{R}_i$ is considered to be *active* during the time interval that begins with its issuance and ends with its completion. Whenever job J_i holds any resources, it is said to be executing within a *critical section*. We let L_i denote the maximum duration of a critical section of J_i and define $L_{max} = \max_{1 \le i \le n}\{L_i\}$.

We allow requests to be nested. The essence of nesting is that jobs are allowed to hold multiple resources simultaneously. Ordinarily, nesting is realized by allowing jobs to request different resources individually. Instead, we assume that such a job requests all of its needed resources via one request. The resulting functionality is equivalent to a mechanism called a *dynamic group lock* (*DGL*) [11], which allows groups of resources to be coalesced under one lock dynamically at runtime. (This is different from ordinary group locks, which are used to coordinate access to groups of resources that are statically determined offline.)

The usage of DGLs avoids deadlock. Another way to avoid deadlock is by requiring resources to be acquired according to some prescribed ordering. When using DGLs instead of this approach, jobs may sometimes have to request resources that are not actually needed if conditional code exists. For example, if after acquiring resource ℓ_a, job J_i acquires one of resources ℓ_b and ℓ_c based on some condition, it would have to acquire all three resources via one request. While this functionality may seem to put DGLs at a disadvantage, the usage of DGLs results in the same worst-case pi-blocking bounds (see below) under all existing RNLP variants as when resource orderings are enforced.

Given our focus on reader/writer sharing, we classify resource accesses as either *reads* or *writes*: a resource may be accessed by multiple jobs concurrently for reading but by only one job at a time for writing. If a job requests multiple resources via one request, we assume that all such resources are requested for either reading or writing. Mechanisms for handling mixed requests, comprised of both read and write accesses, have been presented in prior work [11]; our focus is efficiently processing *non-nested* requests.

If a request $\mathcal{R}_i$ is a read (resp., write) request, then we will often use the notation $\mathcal{R}_i^r$ (resp., $\mathcal{R}_i^w$) to emphasize its type. If its type is not relevant, then we will simply use $\mathcal{R}_i$. We let D_i denote the set of resources requested by $\mathcal{R}_i$. Additionally, we denote the maximum critical-section length over all read (resp., write) requests by any task as L^r_{max} (resp., L^w_{max}).

Pi-blocking. When designing a real-time locking protocol, the primary goal is to enable pi-blocking to be bounded. In the multiprocessor case, the precise definition of pi-blocking is subtle as it depends on how waiting is realized (spinning vs. suspension) and on certain analysis assumptions [2]. Due to space constraints, we limit our attention to protocols that use spinning to realize blocking and that are invoked non-preemptively (*i.e.*, a resource-requesting job is non-preemptive for the entire time it is executing code involving the acquisition, use, and release of resources), but suspension-based variants of our fast RW-RNLP can be obtained by slightly altering the spin-based version presented later. (We are nearing the completion of a suspension-based implementation and intend to release it soon.) Non-preemptive execution is an example of a *progress mechanism* [2]: it ensures that lock-holding tasks are not delayed by untimely preemptions and thus make progress. With spin-based waiting, a job can be considered to be *pi-blocked* if it is spinning.[2]

Analysis assumptions. In our analysis of pi-blocking, we consider critical-section lengths and the number of critical sections per job to be constants, and m and n to be variables, as in prior work [2]. If t is the time at which request $\mathcal{R}_i$ is issued, then we define the *contention* C_i of $\mathcal{R}_i$ to be the number of other active requests at time t that require resources in common with $\mathcal{R}_i$. A reader/writer locking protocol ensures *contention-sensitivity* for a request $\mathcal{R}_i$ if the worst-case pi-blocking for $\mathcal{R}_i$ is $O(1)$ if it is a read request, and $O(C_i)$ if it is a write request. These pi-blocking bounds are asymptotically optimal for non-preemptive, spin-based locking protocols [11].

Related work. In recent years, a number of locking protocols have been presented that are asymptotically optimal with respect to pi-blocking. These include RNLP variants [9, 12, 13] that provide *fine-grained* lock nesting, meaning that each resource is protected by its own lock. The only other protocols known to us that provide fine-grained lock nesting are the multiprocessor bandwidth inheritance protocol [7] and MrsP [5]; however, neither is optimal in any sense. To our knowledge, the only existing protocols that distinguish between read and write requests are single-resource phase-fair locks and the RW-RNLP, both discussed next.

Phase-fair locks. Given our focus (non-preemptive spin locks), 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 single-resource requests [4]. As noted earlier, a phase-fair lock utilizes two FIFO queues, one for read requests and one for write requests, and alternates between read phases and write phases. Several possible implementations of phase-fair locks were considered by Brandenburg and Anderson [4]. They found the phase-fair ticket-lock (PF-TL) to be comparable to or better than other phase-fair implementations from the perspective of lock/unlock costs.

The RW-RNLP. As mentioned earlier, the RW-RNLP uses two per-resource FIFO queues, one for read requests and one for write requests. Furthermore, it uses a mechanism called *request entitlement* to orchestrate reader and writer phases; the entitlement rules determine "who" (reader or writer) must concede to "whom": entitled requests do not concede. In Sec. 3, we consider in detail a new variant of the RW-RNLP, which we call the RW-RNLP*, that is useful for our purposes. We carefully explain there the concept of entitlement. The RW-RNLP is actually a family of protocols because waiting can be realized by spinning or suspension and because different mechanisms for dealing with priority inversions are required depending on how tasks are scheduled. For the non-preemptive, spin-based variant of the RW-RNLP (our focus), worst-case pi-blocking is $O(1)$ for read requests and $O(m)$ for write requests. These bounds are asymptotically optimal, assuming contention for write requests is $\Omega(m)$.

[2]A job can be pi-blocked at release by lower-priority jobs executing non-preemptively. By our analysis assumptions, this *release blocking* is asymptotically upper bounded by the maximum spin blocking for any such jobs, so we focus on spin blocking.

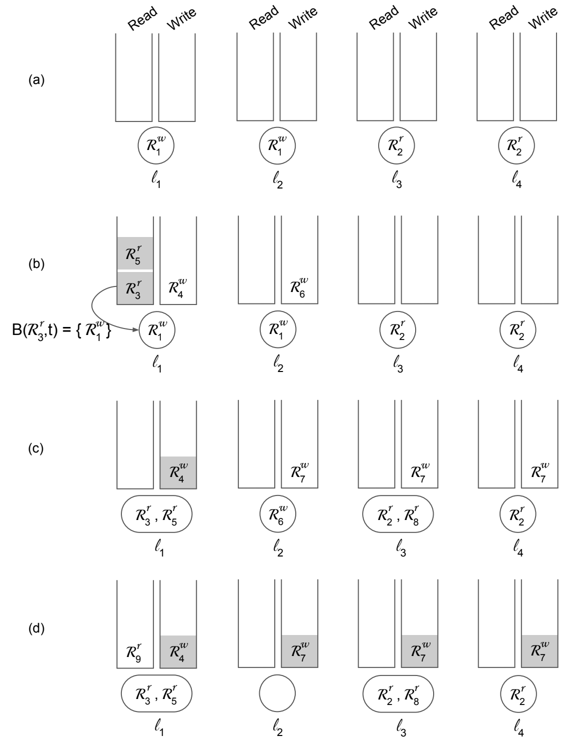


Figure 2: Example illustrating the rules of the RW-RNLP*.

3 THE FAST RW-RNLP

Our proposed fast RW-RNLP is constructed based on a new variant of the RW-RNLP called the RW-RNLP* and existing locking protocols. In this section, we describe the RW-RNLP* and its pi-blocking analysis and then present the fast RW-RNLP.

3.1 The RW-RNLP*

The RW-RNLP* is obtained from the RW-RNLP by altering one aspect of its design and changing the context in which it is applied. 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 3.1. We will use Fig. 2 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 Fig. 2(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.

Basic RW-RNLP rules.* We describe the RW-RNLP* via a set of rules to which an implementation must conform. With the exception of Rule P3, all of the rules below are taken directly from [12]. As we shall see in Sec. 3.2, Rule P3 enables tighter pi-blocking bounds to be computed in our context.

The first three rules place constraints on how the protocol is used; the first two essentially enforce non-preemptive scheduling and the third introduces our specific restricted context.

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.

While Rule P3 may seem restrictive, it will be upheld by definition when the RW-RNLP* is applied in the context of the fast RW-RNLP. It is also trivially upheld in systems with only single-writer resources, which is common use case we consider later. 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.

Example 3.1 (cont'd). Moving from inset (a) to inset (b) in Fig. 2, four additional requests have been issued. Timestamps are determined for these requests when they are issued (Rule G1). (In our examples, jobs issue requests in increasing index order.) 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.

Fig. 2(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. We consider such issues later.

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 [12]). 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 bound pi-blocking times. Below, we use $E(Q_a^w)$ to denote the earliest-timestamped unsatisfied write request for resource ℓ_a.

Example 3.1 (cont'd). In Fig. 2(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 Def. 3.2).[3] (Note that $E(Q_a^w) = \emptyset$ could hold. In this case, we consider $E(Q_a^w) = \emptyset$ to be a "null" request that is not entitled.) $\mathcal{R}_i^r$ remains entitled until it is satisfied.

Example 3.1 (cont'd). In Fig. 2(b), $\mathcal{R}_3^r$ and $\mathcal{R}_5^r$ are both entitled (Def. 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 (Def. 3.2, below). Entitled requests are indicated in Fig. 2 by a light gray shading.

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 Def. 3.1),[3] and ℓ_a is not write locked. $\mathcal{R}_i^w$ remains entitled until it is satisfied.

Example 3.1 (cont'd). In Fig. 2(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 Fig. 2(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. We complete our specification of the RW-RNLP* by stating rules that govern how read and write requests are processed. To state these rules, we introduce notation to allow us identify the set of requests on which an entitled request $\mathcal{R}_i$ (a read or a write) is blocked. Specifically, we let $B(\mathcal{R}_i, t)$ denote the set of requests on which such a request $\mathcal{R}_i$ is blocked at time t.

Example 3.1 (cont'd). In Fig. 2(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 Fig. 2(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 Fig. 2, but $B(\mathcal{R}_i, t)$ 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 3.1 (cont'd). 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 Fig. 2(b). When $\mathcal{R}_1^w$ later completed at some time t, as shown in Fig. 2(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. Fig. 2(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.

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 3.1 (cont'd). When $\mathcal{R}_6^w$ was issued prior to the system state depicted in Fig. 2(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 Fig. 2(c), and at that time t, $B(\mathcal{R}_6^w, t) = \emptyset$ held, so $\mathcal{R}_6^w$ became satisfied, by Rule W2.

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

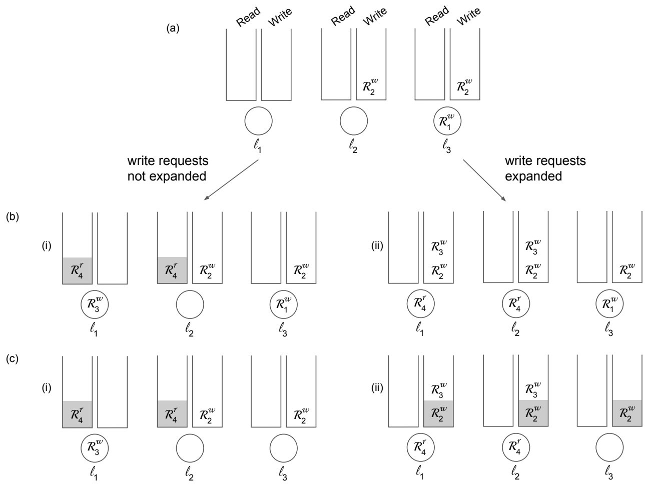


Figure 3: System states without write expansion are labeled (i), and states with write expansion (used in the RW-RNLP) are labeled (ii).

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, we have chosen to avoid introducing the necessary formal machinery to completely define this technique, opting instead for conveying the general idea behind it with an example.

Example 3.2. 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 Fig. 3. 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, though they do not share resources.

Inset (c) shows the situation after $\mathcal{R}_1^w$ has completed. As seen 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 seen in inset (c)(ii), in the scenario with write expansion, the completion of $\mathcal{R}_1^w$ makes $\mathcal{R}_2^w$ entitled.

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 Ex. 3.2 and proven in [12]. Additionally, write expansion eases certain implementation challenges.

In our setting, write expansion is problematic, as our 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 RW-RNLP* Pi-Blocking Bounds

In this section, we derive bounds on the worst-case *acquisition delay* experienced by a request under the RW-RNLP*, *i.e.*, the worst-case time between the issuance and satisfaction of a request. Occasionally, we will find it convenient to distinguish whether a read request $\mathcal{R}_i^r$ or a write request $\mathcal{R}_i^w$ is nested or non-nested. For this purpose, we will use the notation $\mathcal{R}_i^{r,n}$, $\mathcal{R}_i^{r,nn}$, $\mathcal{R}_i^{w,n}$, and $\mathcal{R}_i^{w,nn}$, where the superscript "n" (resp., "nn") means "nested" (resp., "non-nested"). As in [12], we assume that all lock and unlock invocations take no time.

The properties needed to derive acquisition-delay bounds are stated below. Lemma 3.1 and Theorem 3.1 were proved in [12] (appearing as "Lemma 1" and "Theorem 1" there), and those proofs are not affected by the changes we made to the RW-RNLP to obtain the RW-RNLP*. We illustrate each of these properties by referring to our prior example. The remaining properties either require new proofs or are entirely new.

Lemma 3.1. *A write request $\mathcal{R}_i^w$ experiences acquisition delay of at most L_{max}^r time units after becoming entitled.*

Example 3.1 (cont'd). In insets (c) and (d) of Fig. 2, $\mathcal{R}_4^w$ 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 Def. 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 3.1. *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 3.1 (cont'd). Consider $\mathcal{R}_9^r$ in Fig. 2(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 3.2 below is very similar to Lemma 2 in [12] and much of the proof given for it is taken verbatim from there. However, new reasoning is required as we do not employ write expansion.

Lemma 3.2. *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 3.1, it will become satisfied within L_{max}^r time units. Otherwise, by Def. 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 Def. 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^w_{max} time units. At the time t when all such write requests have completed, by Rule R2, each $\mathcal{R}^r_x$ in $B(\mathcal{R}^w_i, t)$ will be satisfied, and by Def. 3.2, $\mathcal{R}^w_i$ will be entitled. By Lemma 3.1, $\mathcal{R}^w_i$ will subsequently experience at most L^r_{max} 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 release.

Corollary 3.1. *If all resources are single-writer resources, then the worst-case acquisition delay of a write request $\mathcal{R}^w_i$ is at most $L^w_{max} + L^r_{max}$ time units.*

Lemma 3.3. *If no nested write requests are active while the non-nested request $\mathcal{R}^{w,nn}_i$ is active, and if $\mathcal{R}^{w,nn}_i$ is the earliest-timestamped active write request for its lone requested resource ℓ_a in D_i, then $\mathcal{R}^{w,nn}_i$ will be satisfied within L^r_{max} time units.*

Proof. The proof of this lemma differs from that given above for Lemma 3.2 only in how Case (ii) in that proof is addressed. For Case (ii) in the context of Lemma 3.3, if the non-nested request $\mathcal{R}^{r,nn}_x$ is entitled, then by Def. 3.1, it must blocked by a satisfied write request $\mathcal{R}^{w,nn}_j$ for resource ℓ_a. However, $\mathcal{R}^{w,nn}_i$ is the earliest-timestamped request for ℓ_a, so Case (ii) is actually impossible in the context of Lemma 3.3. Therefore, $\mathcal{R}^{w,nn}_i$ must be either satisfied or entitled, and in the latter case, it becomes satisfied within L^r_{max} time units, by Lemma 3.1. □

The next two lemmas heavily exploit Rule P3.

Lemma 3.4. *After being issued, a nested write request $\mathcal{R}^{w,n}_i$ will become the earliest-timestamped active write request for all of the resources in D_i within $2L^w_{max} + L^r_{max}$ time units.*

Proof. For any resource in D_i for which $\mathcal{R}^{w,n}_i$ is not the earliest-timestamped write request, by Rule P3, the earliest-timestamped write is a non-nested write request. By Lemma 3.2, each such request is satisfied within $L^w_{max}+L^r_{max}$ time units. By Rule P1, once satisfied, all such non-nested write requests will complete within L^w_{max} time units. Summing these two bounds yields the worst-case bound of $2L^w_{max} + L^r_{max}$ time units stated in the lemma. □

Lemma 3.5. *After being issued, a non-nested write request $\mathcal{R}^{w,nn}_i$ will become the earliest-timestamped active write request for its lone requested resource ℓ_a in D_i:* **(i)** *immediately, if no nested requests are active while $\mathcal{R}^{w,nn}_i$ is active;* **(ii)** *within $4L^w_{max} + 2L^r_{max}$ time units, if nested requests may be active while $\mathcal{R}^{w,nn}_i$ is active.*

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

In Case (ii), if $\mathcal{R}^{w,nn}_i$ is not immediately the earliest-timestamped write request for ℓ_a, then there exists exactly one nested write request $\mathcal{R}^{w,n}_x$ that is the earliest-timestamped write request for ℓ_a. By Lemma 3.4, $\mathcal{R}^{w,n}_x$ will be the earliest-timestamped request for *all* of its requested resources within $2L^w_{max} + L^r_{max}$ time units. By Lemma 3.2, $\mathcal{R}^{w,n}_x$ will be satisfied within an additional $L^w_{max} + L^r_{max}$ time units. Once it is satisfied, by Rule P1, it will complete within L^w_{max} time units. At that time, $\mathcal{R}^{w,nn}_i$ will be the earliest-timestamped write request for its requested resource. Summing all the bounds just stated, this occurs within $4L^w_{max} + 2L^r_{max}$ time units in the worst case. □

Theorem 3.2, given next, provides our desired delay-acquisition bounds. Together with Theorem 3.1, this theorem implies that all pi-blocking bounds under the RW-RNLP* are $O(1)$.

Theorem 3.2. *The worst-case acquisition delay of a write request $\mathcal{R}^w_i$ is:* **(i)** *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;* **(ii)** *$5L^w_{max} + 3L^r_{max}$ 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;* **(iii)** *$3L^w_{max} + 2L^r_{max}$ time units, if $\mathcal{R}^{w,n}_i$ is a nested request.*

Proof. In Case (i), by Lemma 3.5(i), $\mathcal{R}^{w,nn}_i$ will be the earliest-timestamped active write request for its lone requested resource as soon as it is issued. By Lemma 3.3, it will be satisfied within L^r_{max} time units.

In Case (ii), by Lemma 3.5(ii), $\mathcal{R}^{w,nn}_i$ will be the earliest-timestamped active write request for its lone requested resource within $4L^w_{max} + 2L^r_{max}$ time units. By Lemma 3.2, it will then be satisfied within $L^w_{max} + L^r_{max}$ time units, resulting in a worst-case acquisition delay of $5L^w_{max} + 3L^r_{max}$ time units.

In Case (iii), by Lemma 3.4, $\mathcal{R}^{w,n}_i$ will be the earliest-timestamped active write request for all of its requested resources within $2L^w_{max} + L^r_{max}$ time units. By Lemma 3.2, it is then satisfied within $L^w_{max} + L^r_{max}$ time units, resulting in a worst-case acquisition delay of $3L^w_{max} + 2L^r_{max}$ time units. □

It can be shown that all of the blocking bounds in Theorem 3.2 are *tight, i.e.*, scenarios exist in which these exact bounds occur.[4] Notice that, by Theorem 3.1 and Theorem 3.2(i), if non-nested requests are not affected by nested requests, then read and write requests have worst-case pi-blocking bounds of only $L^w_{max} + L^r_{max}$ and L^r_{max} time units, respectively.

3.3 Putting the Pieces Together

In this section, we describe our proposed fast RW-RNLP protocol. Our goals for this protocol are threefold: **(i)** non-nested requests should have low lock/unlock overheads; **(ii)** such requests should have contention-sensitive worst-case pi-blocking bounds; **(iii)** nested requests should have worst-case pi-blocking bounds that are asymptotically the same as under the RW-RNLP. In describing the fast RW-RNLP below, we verify that Goals (ii) and (iii) are met. We address Goal (i) later when we discuss an implementation of the protocol and an experimental evaluation of that implementation.

The fast RW-RNLP is defined by using the lock and unlock routines of the RNLP, the RW-RNLP*, and ordinary (not phase-fair) mutex ticket locks (TLs) [10] as subroutines, as shown in Fig. 4.[5] Recall that the RNLP provides mutex sharing and supports nested requests. Under it, the worst-case pi-blocking of any request is $O(m)$ [13]. A TL provides mutex sharing for a single resource and ensures contention-sensitive pi-blocking.

Referring to the fast RW-RNLP structure in Fig. 4, notice that all read requests (both nested and non-nested) directly invoke the RW-RNLP*. Furthermore, any nested write request that requires access to a resource ℓ_a must first "acquire" that resource within the context of the RNLP and then invoke the RW-RNLP*. Also, any non-nested write request for that resource must first "acquire" that resource within the context of a TL associated with that resource and then

[4] See online appendix: http://www.cs.unc.edu/anderson/papers.html.

[5] The lock and unlock routines for the 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 RW-RNLP*.

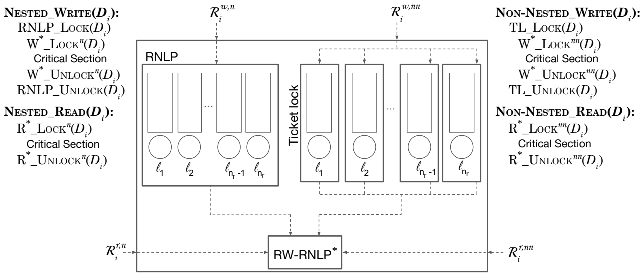


Figure 4: Fast RW-RNLP structure.

invoke the RW-RNLP*. This overall protocol structure ensures that Rule P3 is upheld from the perspective of the RW-RNLP*.

Because read requests directly invoke the RW-RNLP*, by Theorem 3.1, the pi-blocking incurred by them is $O(1)$ in the worst case (we consider L_{max} to be constant). Thus, Goals (ii) and (iii) above are met for read requests. The following theorem shows that these goals are also met for write requests; the pi-blocking incurred by a non-nested write request $\mathcal{R}_i^{w,nn}$ is $O(C_i)$ in the worst case (recall that C_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. (Referring to Goal (iii), we note that the worst-case pi-blocking for write requests under the RW-RNLP is $O(m)$ [12].[6])

THEOREM 3.3. *Under the fast RW-RNLP, the worst-case acquisition delay for a write request $\mathcal{R}_i^w$ is:* **(i)** $C_i(L^w_{max}+L^r_{max})+L^r_{max}$ *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)** $C_i\cdot(6L^w_{max}+3L^r_{max})+5L^w_{max}+3L^r_{max}$ *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;* **(iii)** $(m-1)\cdot(4L^w_{max}+2L^r_{max})+3L^w_{max}+2L^r_{max}$ *time units, if $\mathcal{R}_i^{w,n}$ is a nested request.*

PROOF. In Case (i), $\mathcal{R}_i^{w,nn}$ must wait for up to C_i contending write requests ahead of it in the TL associated with its lone requested resource. By Theorem 3.2(i), each of these write requests may face an acquisition delay of up to L^r_{max} time units within the RW-RNLP* and then execute its critical section for up to L^w_{max} time units. Thus, within $C_i\cdot(L^w_{max}+L^r_{max})$ time units after being issued, $\mathcal{R}_i^{w,nn}$ will not be blocked by any write requests in the TL associated with its requested resource. At that time, $\mathcal{R}_i^{w,nn}$ will invoke the RW-RNLP* and, again by Theorem 3.2(i), experience an acquisition delay of up to L^r_{max} time units. In total, this yields a worst-case acquisition delay of $C_i\cdot(L^w_{max}+L^r_{max})+L^r_{max}$ time units for $\mathcal{R}_i^{w,nn}$.

Case (ii) is similar to Case (i) except that Theorem 3.2(ii) is applied instead of Theorem 3.2(i). Thus, the worst-case acquisition delay is $C_i\cdot(6L^w_{max}+3L^r_{max})+5L^w_{max}+3L^r_{max}$ time units.

In Case (iii), $\mathcal{R}_i^{w,n}$ must wait within the RNLP for up to $m-1$ other requests to complete before it can invoke the RW-RNLP*. Arguing as in the cases above, but this time using Theorem 3.2(iii), a worst-case acquisition delay of $(m-1)\cdot(4L^w_{max}+2L^r_{max})+3L^w_{max}+2L^r_{max}$ time units results. □

Note that, in a system with only single-writer resources, the RW-RNLP* alone is sufficient and Cor. 3.1 can be applied to show that all requests incur $O(1)$ pi-blocking with very low constant factors.

To this point, we have fully specified the RW-RNLP* abstractly. What remains is to devise an actual implementation of it with reasonable overheads. We consider this issue next.

[6]More precisely, the bound presented is $(m-1)(L^w_{max}+L^r_{max})$.

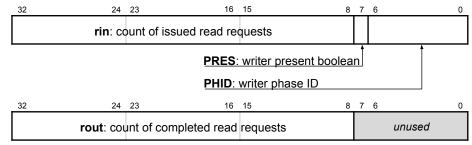


Figure 5: Bits in the per-resource *rin* and *rout* variables. (A very similar figure appears in [4].)

4 IMPLEMENTATION AND EVALUATION

Of the building blocks used to construct the fast RW-RNLP, the TL and the RNLP have existing implementations [4, 12]. Therefore, it remains for us to provide an implementation of the RW-RNLP* as well as an experimental evaluation of the overall fast RW-RNLP. Recall that we focus on the user-level, spin-based version.

4.1 Implementation

The main challenge in implementing the RW-RNLP* lies in supporting the atomicity assumptions inherent in the rule-based specification of it. 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 [12]. While such an approach introduces additional pi-blocking, the protected critical sections are usually very short, so we consider such blocking to be part of the lock and unlock overhead of the protocol being implemented. Still, we would like to avoid relying on the use of mutex protocols in this way if possible, and we want to *categorically preclude* their use in implementing the lock and unlock routines for non-nested requests, as efficiently implementing such routines is the emphasis of this paper.

With these concerns in mind, we now describe our implementation of the RW-RNLP*.

Shared variables of the RW-RNLP.* From the point of view of our implementation, each shared resource ℓ_a is viewed as pointer to a structure called *res_state*, which consists of four shared counters, *rin*, *rout*, *win*, and *wout*, as shown in Listing 1. Almost identical counters to these are used in the PF-TL [4]. 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 Fig. 5. Listing 1 shows various constant bit vectors used in our code to access and manipulate certain bits in *rin* and *rout*.

Listing 1 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
```

Non-nested requests in the RW-RNLP.* The lock and unlock routines for non-nested requests in our implementation are shown in Listing 2. These are *nearly identical to those for the PF-TL* [4], which to our knowledge is *the most efficient reader/writer lock for single-resource requests proposed to date.* A non-nested read $\mathcal{R}_i^{r,nn}$ of 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 a 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 9–10). 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 11–12). 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).

Listing 2 RW-RNLP* Routines for Non-Nested Reqs.

```
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:  procedure R*_Unlock^nn(ℓ: ptr to res_state)
6:      atomic_add(ℓ→rout, RINC)
7:  procedure W*_Lock^nn(ℓ: ptr to res_state)
8:      var rticket, wticket, w: unsigned int
9:      wticket := fetch&add(ℓ→win, 1)                         ▷ In write queue
10:     await (wticket = ℓ→wout)                               ▷ Head of write queue
11:     w := PRES | (wticket & PHID)
12:     rticket := fetch&add(ℓ→rin, w)
                                    ▷ Marked entitled now for all reads to see
13:     await (rticket = ℓ→rout)                               ▷ Satisfied
14: procedure W*_Unlock^nn(ℓ: ptr to res_state)
15:     fetch&and(ℓ→rin, 0xFFFFFF00)                           ▷ Clear WBITS
16:     ℓ→wout := ℓ→wout + 1
```

Nested requests in the RW-RNLP.* The lock and unlock routines for nested requests are shown in Listing 3. These routines are very similar to those in Listing 2, with two notable exceptions. First, an extra phase has been added to the lock routine for read requests (Lines 19–21). Introducing this extra phase eliminates unnecessary writer blocking in one particular corner case.[4] Second, because requests are now for *sets* of resources, we need to ensure that such sets can be enqueued atomically to prevent potential deadlock. (This is why, as discussed in Sec. 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, we discovered that this race condition can be eliminated by requiring each nested read request to hold a global PF-TL for *writing* when updating multiple read queues (Lines 22–25) and by requiring each nested write request to hold this PF-TL for *reading* when it updates multiple write queues (Lines 36–40). (The calls to the phase-fair lock and unlock routines in Lines 22, 25, 36, and 40 do not specify input parameters because we have no need to distinguish different shared resources protected by these routines.) While using a PF-TL introduces blocking overhead, this overhead is only $O(1)$ for write requests, which require only read access. This is preferable to the blocking overhead that would result from using a mutex lock.

Clearly, the routines in our implementation are not actually atomic: each executes over durations of time, not instantaneously. However, it can be formally shown that each routine is linearizable.[4] That is, for each routine, an instantaneous *linearization point* can be defined at which the routine "appears" to take effect atomically. When viewing these routines in this way, they can be shown to support the rule-based specification of the RW-RNLP* given earlier.

Listing 3 RW-RNLP* Routines for Nested Reqs.

```
17: procedure R*_Lock^n(D: set of ptr to res_state)
18:     var w_ℓ: unsigned int for each ℓ in D
19:     for each ℓ in D:
20:         w_ℓ := ℓ→rin & WBITS
21:         await (w_ℓ = 0) or (w_ℓ ≠ (ℓ→rin & WBITS))
22:     PFTL_W_Lock()                                          ▷ Write-lock global PFTL
23:     for each ℓ in D:
24:         w_ℓ := fetch&add(ℓ→rin, RINC) & WBITS
25:     PFTL_W_Unlock()                                        ▷ Unlock global PFTL
26:     for each ℓ in D:
27:         await (w_ℓ = 0) or (w_ℓ ≠ (ℓ→rin & WBITS))         ▷ Satisfied
28: procedure R*_Unlock^n(D: set of ptr to res_state)
29:     for each ℓ in D:
30:         atomic_add(ℓ→rout, RINC)
31: procedure W*_Lock^n(D: set of ptr to res_state)
32:     var rticket_ℓ, wticket_ℓ, w_ℓ: unsigned int for each ℓ in D
33:     for each ℓ in D:
34:         wticket_ℓ := fetch&add(ℓ→win, 1)                   ▷ In write queue
35:         await (wticket_ℓ = ℓ→wout)
                                    ▷ Head of all requested write queues now
36:     PFTL_R_Lock()                                          ▷ Read-lock global PFTL
37:     for each ℓ in D:
38:         w_ℓ := PRES | (wticket_ℓ & PHID)
39:         rticket_ℓ := fetch&add(ℓ→rin, w_ℓ)
                                    ▷ Marked entitled now for all reads to see
40:     PFTL_R_Unlock()                                        ▷ Unlock global PFTL
41:     for each ℓ in D:
42:         await (rticket_ℓ = ℓ→rout)                         ▷ Satisfied
43: procedure W*_Unlock^n(D: set of ptr to res_state)
44:     for each ℓ in D:
45:         fetch&and(ℓ→rin, 0xFFFFFF00)                       ▷ Clear WBITS
46:         ℓ→wout := ℓ→wout + 1
```

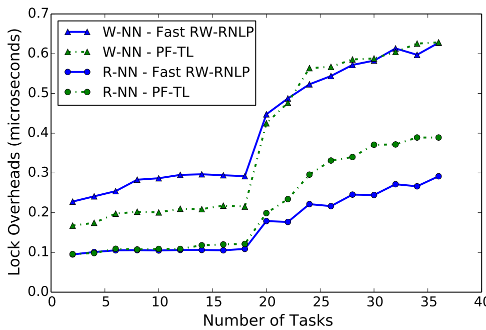
4.2 Evaluation

We conducted a user-space experimental evaluation of the fast RW-RNLP in which lock/unlock overheads and observed blocking times were recorded under a variety of scenarios. Given the focus of this paper, we were particularly interested in overheads and blocking times for non-nested requests. We conducted our experiments on a dual-socket, 18-cores-per-socket Intel Xeon E5-2699 platform.

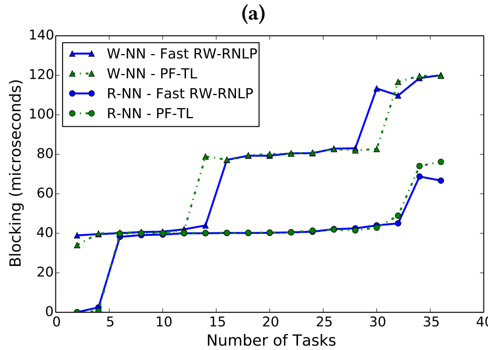
In our experiments, we varied a number of experimental parameters including the numbers of tasks and resources, nesting depths and critical-section lengths of requests, and ratios of non-nested to nested requests and of read to write requests. Each task was pinned to a single core, and for task counts of up to 18, all tasks were assigned to the same socket. Each task was configured to issue lock and unlock calls 1,000 times to simulate behavior that would generate the worst-case lock overhead and blocking times. In all of our graphs, we plot these worst-case values, which were obtained by computing the 99$^{\text{th}}$ percentile of all recorded results in order to filter out any spurious measurements (our measurements were taken at user level, so we have no other means for filtering results impacted by interrupts).

Overheads and blocking. We compared the considered protocols on the basis of overhead and blocking: 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. We measured both overhead and blocking for a number of different scenarios. Each such scenario was defined by specifying particular values or ranges for the experimental parameters mentioned above.

In designing the fast RW-RNLP, we have sought to ensure that **(i)** non-nested requests have low overhead and experience contention-sensitive pi-blocking and **(ii)** nested requests experience pi-blocking

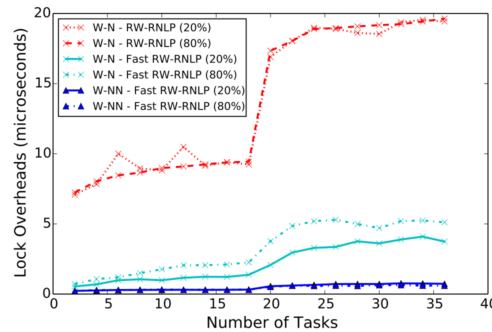


(a)

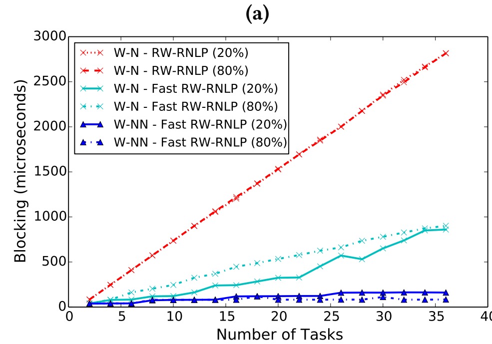


(b)

Figure 6: (a) Lock overheads and (b) blocking for non-nested read and write requests when using PF-TLs versus 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.



(a)



(b)

Figure 7: (a) Overhead and (b) 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% and 80% of requests being nested. Due to write expansion (recall Fig. 3), D_i was inflated to include all 64 resources for writes under the RW-RNLP.

that is no worse (and hopefully better) than that under the RW-RNLP. Accordingly, as standards for comparison, we considered the use of per-resource PF-TLs (which exhibit very low overhead and are contention-sensitive) in assessing (i) and the RW-RNLP (of course) in assessing (ii). In the course of our experiments, we produced hundreds of graphs. The full set of graphs can be found online.[4] A few graphs that are exemplars of trends seen generally are discussed in the following observations.

Obs. 1. *For non-nested read and write requests, the fast RW-RNLP and PF-TLs exhibited comparable overheads.*

This observation is supported by Fig. 6(a), which plots lock overheads 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 reads and writes, 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 $40\mu s$. For comparison, lock overheads for both protocols hold steady in the range of around $1.0\mu s$ to $2.5\mu s$ for up to 18 tasks, with the fast RW-RNLP having a slightly higher write-lock overhead than PF-TLs. Beyond 18 tasks, lock overheads increase under both protocols. This is because, beyond a task count of 18, tasks are executing on both sockets of the considered platform. Notice that, beyond a task count of 18, the write-lock overheads of both protocols converge, and the read-lock overhead of the fast RW-RNLP becomes slightly better. We suspect that the better read-lock overhead of the fast RW-RNLP is due to reduced cache invalidations of shared lock state caused by contending write requests, which must first acquire a ticket lock under the fast RW-RNLP. We omit graphs showing unlock overheads due to space constraints, but they showed similar trends.

Obs. 2. *In general, overheads increased when using two sockets instead of one.*

This trend is seen in Fig. 6(a), discussed earlier, and also in Fig. 7(a), considered in detail below. When tasks execute on two sockets instead of one, overheads due to maintaining cache coherency increase. Observe that, in Fig. 6(a), lock overheads under the fast RW-RNLP are never more than around $0.6\mu s$. This value is quite small compared to the $40\mu s$ critical-section length. Note that any blocking is mostly a function of critical-section lengths.

Obs. 3. *In scenarios with only non-nested requests, the fast RW-RNLP and PF-TLs exhibited nearly identical blocking.*

This observation is clearly supported by Fig. 6(b). Together with Obs. 1, this observation suggests the viability of providing the fast RW-RNLP as a general synchronization solution. It can even be used in systems in which nested requests do not occur with no detrimental impacts of note.

Obs. 4. *In scenarios with both nested and non-nested requests, overheads for write requests tended to be much lower under the fast RW-RNLP than under the RW-RNLP.*

This observation is supported by Fig. 7(a), which depicts data from two different scenarios as detailed in the figure's caption. The higher overheads under the RW-RNLP are partially due to the use of write expansion (recall Fig. 3), which increases resource contention. This increased contention impacts the overhead of write requests, as they write-lock an underlying PF-TL to update all relevant resource queues atomically. Note that, under the RW-RNLP, write expansion forces non-nested write requests to be processed like nested ones.

Notice that Fig. 7 pertains to write requests. The corresponding read request results at $m = 36$ show overheads of around $0.3\mu s$ for non-nested requests under the fast RW-RNLP compared to around $0.8\mu s$ under the RW-RNLP. Under the fast RW-RNLP, non-nested requests had higher blocking by about one critical-section length, and nested read requests had higher overhead (of around $3\mu s$) and higher blocking by a few critical-section lengths.

Obs. 5. *In scenarios with both nested and non-nested requests, blocking for write requests tended to be much lower under the fast RW-RNLP than under the RW-RNLP.*

This observation is supported by Fig. 7(b), which plots recorded worst-case blocking times associated with the scenarios in Fig. 7(a). For $m = 36$, blocking was 17 times lower under the fast RW-RNLP than under the RW-RNLP; write expansion increases resource contention, which increases blocking times of the RW-RNLP.

Obs. 6. *Non-nested requests exhibited contention-sensitive blocking under the fast RW-RNLP but not the RW-RNLP.*

This observation is also supported by Fig. 7(b). 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. 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 $O(C)$.

Of relevance to the analysis presented in Sec. 3, Fig. 8 demonstrates the results of varying the critical-section length while holding the number of tasks n constant (in our experiments, m and n are equal). In contrast, in Fig. 7(b) the number of tasks was varied, and the critical-section length was held constant; the points in Fig. 7(b) at $m = 36$ are the same as those in Fig. 8 for $L_i = 40\mu s$. Note that varying m effectively modifies the term C_i for each request $\mathcal{R}_i$.

Obs. 7. *Blocking time scaled linearly with critical-section length for both the fast RW-RNLP and the RW-RNLP.*

Fig. 8 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.

Although our approach results in higher coefficients for the nested write requests than the bounds proven for the RW-RNLP, lower blocking times were generally seen under the fast RW-RNLP. We suspect this difference is because, under the RW-RNLP, write expansion guarantees that all write requests conflict.

We also noted differences between nested and non-nested write requests under the fast RW-RNLP, highlighting the improvement of $O(C)$ over $O(m)$ blocking. Under the fast RW-RNLP, the $O(C)$ 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.

5 CONCLUSION

We have presented a new RNLP variant, the fast RW-RNLP, which employs a fast-path mechanism to provide contention-sensitive pi-blocking and low processing costs for non-nested lock requests, while preserving the RW-RNLP's asymptotic pi-blocking bounds for

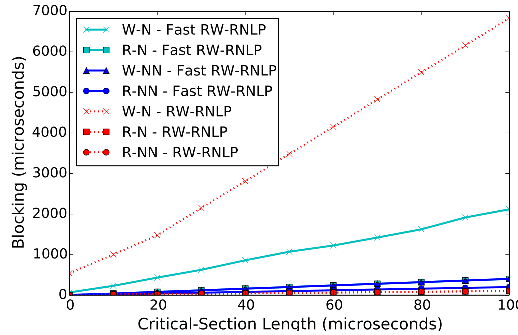


Figure 8: Blocking for nested and non-nested write requests under the RW-RNLP and the fast RW-RNLP. The critical-section length varies, $m = 36$, $n_r = 64$, $|D_i| = 1$, for non-nested requests, and $|D_i| = 4$, for nested requests. ($|D_i|$ is inflated to 64 under the RW-RNLP as above.) A request was chosen to be a write with probability 0.5.

nested requests. While the goal of ensuring contention sensitivity *efficiently* in the general case (nested requests) has so far proven to be elusive, we have shown that it is at least possible to do so for the common case of non-nested requests even when nested requests exist. To ensure contention-sensitivity for non-nested requests, we had to eliminate the write-expansion rule of the RW-RNLP. In our experiments, this had a positive impact on blocking for all requests.

The fast RW-RNLP has a modular structure that enables different variants to be applied in different contexts. For example, the RW-RNLP* gives constant-time access to all resource requests in systems comprised of single-writer, multiple-reader resources. Additionally, the RNLP component in Fig. 4 could be replaced by the C-RNLP to obtain contention-sensitive pi-blocking for nested requests (at the expense of higher overheads for such requests). Further variants realize task waiting by suspending tasks rather than by requiring them to block by spinning; the implementation of one such variant is in progress. In a future expanded version of this paper, we will discuss these variants in full. We plan to compare the fast RW-RNLP to other alternatives by conducting a large-scale overhead-aware schedulability study. Such a study will allow us to assess the extent to which the more efficient processing of non-nested requests affects the ability to ensure timing correctness in a holistic sense.

REFERENCES

[1] D. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for java. In *PLDI 1998*.

[2] B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, University of North Carolina, Chapel Hill, NC, 2011.

[3] B. Brandenburg and J. Anderson. Feather-trace: A lightweight event tracing toolkit. In *OSPERT 2007*.

[4] B. Brandenburg and J. Anderson. Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-Time Systems*, 46(1), 2010.

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

[6] P. Courtois, F. Heymans, and D. Parnas. Concurrent control with readers and writers. *CACM*, 14(10), 1971.

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

[8] H. Huang, P. Pillai, and K. Shin. Improving wait-free algorithms for interprocess communication in embedded real-time systems. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, 2002.

[9] C. Jarrett, B. Ward, and J. Anderson. A contention-sensitive fine-grained locking protocol for multiprocessor real-time systems. In *RTNS 2015*.

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

[11] B. Ward. *Sharing Non-Processor Resources in Multiprocessor Real-Time Systems*. PhD thesis, University of North Carolina, Chapel Hill, NC, 2016.

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

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