Noname manuscript No.
(will be inserted by the editor)

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

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

Received: date / Accepted: date

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 18 times lower than under an existing RNLP variant and improved schedulability over that variant and a simple group lock.

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

---

Work supported by NSF grants CNS 1409175, CPS 1446631, CNS 1563845, and CNS 1717589, AFOSR grant FA9550-14-1-0161, ARO grant W911NF-14-1-0499, ARO grant W911NF-17-1-0294, and funding from General Motors. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGS 1650116. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

C. E. Nemitz
UNC-Chapel Hill, 201 S. Columbia St., Chapel Hill, NC 27599
E-mail: nemitz@cs.unc.edu

T. Amert
UNC-Chapel Hill, 201 S. Columbia St., Chapel Hill, NC 27599
E-mail: tamert@cs.unc.edu

J. H. Anderson
UNC-Chapel Hill, 201 S. Columbia St., Chapel Hill, NC 27599
E-mail: anderson@cs.unc.edu

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 [7, 14]. 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 [50], which provides mutex sharing, the RW-RNLP [52], which provides reader/writer sharing, and the C-RNLP [34], 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 an overlapping set of resources [34]. 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.

Transitive Blocking. Under any non-contention-sensitive RNLP variant, requests may become part of transitive blocking chains caused by nested requests. These chains cause requests, even non-nested requests, to have non-contention-sensitive pi-blocking bounds. A simple example is given in Fig. 1, which depicts two resources ℓ_a and ℓ_b, on m processors, accessed by m 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 FIFO-ordered locking protocol, such as a ticket lock. In this case, $\mathcal{R}_m$ 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 a non-contention-sensitive RNLP variant 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.

The price of supporting nested requests. 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

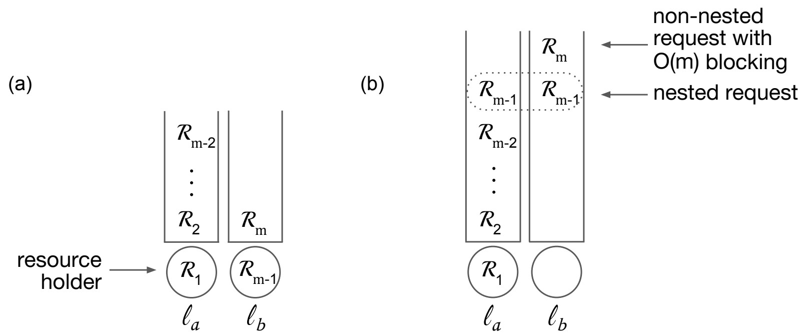

Fig. 1: Impact of transitive blocking on non-nested requests. In (b), $\mathcal{R}_{m-1}$ requests ℓ_a and ℓ_b together using a dynamic group lock (defined in Sec. 2), as allowed by the RNLP and RW-RNLP.

ensuring non-nested lock requests **(i)** are contention-sensitive and **(ii)** incur low lock/unlock overhead comparable to that 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 [23]. (Since reader/writer sharing subsumes mutex sharing, the fast RW-RNLP can be applied to support the latter.) A preliminary version of this work presented the fast RW-RNLP, which uses a component called the RW-RNLP* [40]. In this work we present a second variant with the reader-reader-reader phase-fair locking protocol (R^{3}LP) as a central component; the R^{3}LP extends the notion of a *reader-only phase-fair lock* [41]. We derive tighter bounds for the fast RW-RNLP when the R^{3}LP is used in place of the RW-RNLP*. Additionally, we show the schedulability benefits of the fast RW-RNLP with the R^{3}LP by presenting a large-scale schedulability study that compares the two fast RW-RNLP variants to existing protocols.

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 [17]. 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 overhead (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

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

time. This affects the queueing logic and the orchestration of phases. The latter becomes more difficult as different resources may be in different phases.

In our 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 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 [33]. Additionally, by altering one of the components, contention-sensitivity can be ensured for nested requests (like with the C-RNLP, but at the expense of greater overhead for such requests). Waiting in the fast RW-RNLP can be realized by either spinning or suspension, though we consider only the former in detail in this work. 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 is borne out in experiments we conducted in which lock/unlock overhead and observed pi-blocking times were recorded for non-nested requests. We found that lock/unlock overhead for such requests is nearly identical under the fast RW-RNLP and PF-TLs. We also found that 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 schedulability study we present, in which 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.

Organization. In the rest of the paper, we give needed background and discuss related work (Sec. 2), describe the fast RW-RNLP in detail along with two new protocols that are applied as components within it (Secs. 3 and 4), discuss our experimental results (Sec. 5), and conclude (Sec. 6).

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.*, a partitioned, global, or clustered earliest-deadline-first scheduler). 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. We say that a locking protocol is *fine-grained* if each resource is protected individually.

Ordinarily, nesting is realized by allowing jobs to request and acquire each resource individually in a sequential fashion. This approach can substantially inflate critical-section lengths, as the first resource is acquired before a second is requested. Any blocking a job may experience while waiting for a second resource must be counted toward the critical-section length of the first resource [46]. To prevent deadlock when using this approach, requests must acquire resources according to some prescribed ordering [24, 31]. Locking protocols that use this approach are fine grained, however, which can allow tighter blocking bounds to be computed in some cases.

A second approach to handling nested lock requests is to statically group resources in a manner that eliminates nesting. This coarse-grained approach allows protocols to be used that do not otherwise allow nesting, at the cost of reduced parallelism. By the way resources are defined (as groups of resources), protocols controlling access to these resources cannot deadlock.

To avoid critical-section inflation and minimize lost parallelism, we instead assume that 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*) [49], which allows groups of resources to be coalesced under one lock dynamically at runtime. (This is different from the ordinary group locks described above, which are used to coordinate access to groups of resources that are statically determined offline.) The usage of DGLs also avoids deadlock. When using DGLs, 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. This approach is also fine grained, as resources can be added to the dynamic group individually.

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 [49]; 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 $\mathcal{R}_i$. 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"). 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 [12]. 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 herein. Non-preemptive execution is an example of a *progress mechanism* [12]: 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 [12]. 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 [49].

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 non-nested requests [17]. Several possible implementations of phase-fair locks were considered by Brandenburg and Anderson [17]. 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. We extend the concept of a phase-fair lock to a general phase-fair reader-only lock in Sec. 3.

[2] A job can also 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.

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. 4, 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 if contention for write requests is $\Omega(m)$.

Other related work. As the prevalence of multiprocessor systems has grown, the literature on support for shared resources in a multiprocessor context has grown accordingly [2–6, 10, 11, 13, 12, 15, 16, 18, 17, 19–22, 25–30, 34, 37, 39–44, 46–49, 51, 52, 50, 53–57]. However, much of this work does not handle nested resource accesses. We highlight some of these approaches that do not handle nested requests (except by means of a static, coarse-grained group lock) before covering prior work that allows fine-grained lock nesting.

Several multiprocessor protocols expanded on their uniprocessor counterparts. The distributed and multiprocessor priority ceiling protocols (DPCP and MPCP) [42–44] build on ideas of the priority ceiling protocol (PCP) [45]. The MPCP was then used as a basis for developing a partitioning heuristic for resource-sharing tasks [37]. The multiprocessor stack resource protocol expands on the ideas of the stack resource protocol [8] by classifying resources as local or global; access to global resources is coordinated in FIFO order, with waiting processes spinning non-preemptively [29]. This was then compared to the MPCP [28]. Beyond multiprocessor ceiling-based protocols [22], the FMLP [11], FMLP+ [13], and OMLP [19] have been developed. Many of the above protocols have been implemented and those implementations compared [12, 15, 16].

Other work explored resource-sharing schemes beyond mutual exclusion. For example, k-exclusion protocols [18, 25, 53] and new resource-sharing paradigms like preemptive mutual exclusion and half-protected sharing [48] have been developed. Prior work has also investigated different priorities at which a blocked task may spin; at lower priorities, spinning tasks may suspend [2–4]. Work has also been done to coordinate resource sharing between independently developed system components that are then used modularly in a larger system [5, 6, 39].

Within the context of various protocols that grant access to resources in a non-nested fashion, computing reasonably tight blocking bounds has also 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 [21]. 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 [54]. Comparing global semaphore protocols based on six types of delay was also enabled by linear optimization [56].

Challenges with allowing arbitrary nesting have long been known. Imposing a constraint that nesting depth be at most two allowed for the development of FIFO- and priority-inheritance-based spin-lock algorithms [46,47]. Accurately bounding the blocking caused by arbitrarily nested resource access is NP-hard [55]. A graph-based abstraction has been presented that serves as the basis for an integer linear program to compute blocking [10].

In recent years, a number of locking protocols have been presented that are asymptotically optimal with respect to pi-blocking. These include RNLP variants [34,51,52,50] that provide fine-grained lock nesting. The only other protocols known to us that provide fine-grained lock nesting are the multiprocessor bandwidth inheritance protocol (M-BWI) [26,27] and MrsP [20,30,57]; however, neither is optimal in any sense. (Both the M-BWI and MrsP handle nested requests by requiring resources to be acquired sequentially.) Only approaches that are explicitly contention-sensitive [34,40,41] can reduce the worst-case blocking analysis from $O(m)$ for nested requests due to the challenge shown in Fig. 1; all other approaches to handling nested requests may encounter such a chain of blocking.

The M-BWI expands the original uniprocessor protocol, which allows a resource-holding task to inherit the execution budget or bandwidth of a higher-priority task, to work in a multiprocessor context. This is accomplished by reasoning about the resource reservation servers that grant bandwidth and coordinating between such servers on different processors. The M-BWI has been implemented and compared to the FMLP and OMLP on the basis of schedulability [27].

MrsP 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 [30].

3 Reader-Only Phase-Fair Locks

The main contribution of this paper is the fast RW-RNLP, which is presented in Sec. 4. Two components of the fast RW-RNLP employ *reader-only phase-fair locks*, a notion first introduced by us in restricted form in a recent paper [41]. In this section, we review the basic mechanisms of reader-writer phase-fair locks before introducing reader-only phase-fair locks and a corresponding implementation. We also present bounds on the worst-case *acquisition delay*, *i.e.*, the worst-case time between the issuance and satisfaction of a request, experienced by a request under several variants of phase-fair locks. As in [52], we assume that all lock and unlock invocations take no time.

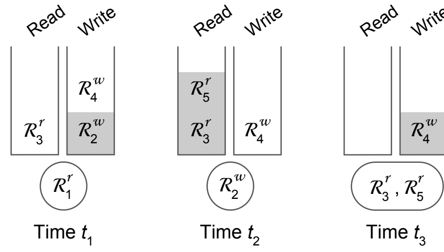

Fig. 2: Illustration of reader-writer phase-fair locking protocol managing access to a resource.

3.1 Reader-Writer Phase-Fair Locks

As described in Sec. 1, 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. In the presence of both types of requests, all active read requests are satisfied at the start of a read phase and one active write request is satisfied at the start of a write phase.

In Fig. 2 and subsequent figures, we use gray shading to indicate which requests will execute in the next phase; this shading corresponds to the notion of *request entitlement*, as described in Sec. 2 and defined formally in Sec. 4. A formal definition is not necessary for a basic understanding of the protocols under consideration. In our examples, jobs issue requests in increasing index order; Ex. 1, depicted in Fig. 2, is comprised of five requests, $\mathcal{R}_1$ through $\mathcal{R}_5$.

Example 1 Fig. 2 depicts the state of a reader-writer phase-fair 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. 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.

3.2 Reader-Reader Phase-Fair Locks

In recent work, we introduced the reader-reader phase-fair locking protocol, which we denote as R^{2}LP [41]. The R^{2}LP arbitrates access to a resource between two types of read requests; an arbitrary read request of Type 1 (resp., Type 2) may execute concurrently with requests of the same type but may not execute with requests of Type 2 (resp., Type 1). (The problem of supporting multiple types of read requests is similar to the group mutual exclusion problem [35, 36] except that we require $O(1)$ pi-blocking bounds.)

We denote a read request of Type 1 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

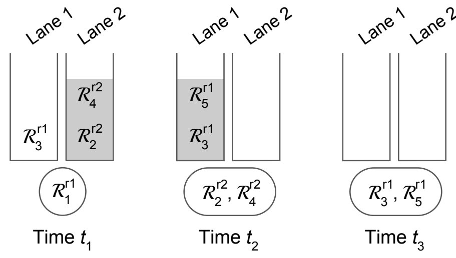

Fig. 3: R^2LP illustration with read requests of Type 1 and Type 2.

type. For example, $\mathcal{R}_3^{r1}$ in Fig. 3 is enqueued in Lane 1. We now walk through the phase transitions of the R^2LP in an example.

Example 2 As shown in Fig. 3, 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^2LP 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.

The above example illustrates the basic functionality of the R^2LP, which was originally presented in [41]. We state without proof the following lemma, which is a generalization of a corresponding result presented there (as Theorem 5). In this lemma, we use L^{r1}_{max} (resp., L^{r2}_{max}) to indicate the maximum critical-section length of a read request of Type 1 (resp., Type 2).

Lemma 1 *Under the R^2LP, the worst-case acquisition delay of a request of any type is $L^{r1}_{max} + L^{r2}_{max}$ time units.*

3.3 Reader-Reader-Reader Phase-Fair Locks

In this paper, we introduce the reader-reader-reader phase-fair locking protocol (R^3LP). The R^3LP 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 As shown in Fig. 4, 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 the following phase, $\mathcal{R}_5^{r2}$ will be satisfied, as indicated with the light gray shading. As with the R^2LP, under the R^3LP, $\mathcal{R}_3^{r1}$ cannot be satisfied at t_1.

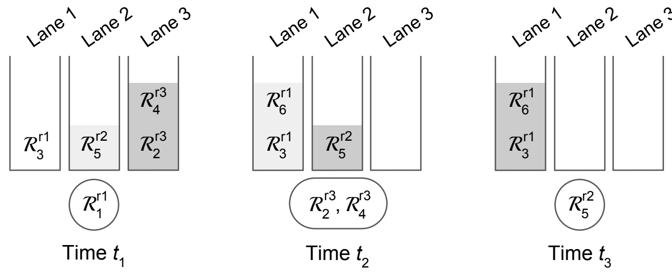

Fig. 4: R^{3}LP illustration with read requests of Type 1, Type 2, and Type 3.

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.

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.

3.4 R^{3}LP Implementation

Our implementation of the R^{3}LP expands on concepts from the R^{2}LP. We start by introducing the shared variables. Then we present the pseudocode for requests of Type 1.

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
```

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, we define a set of variables 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

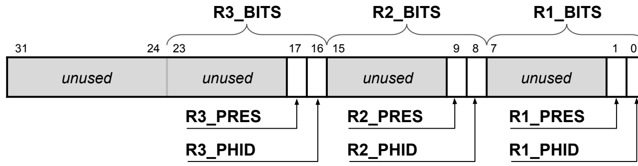

Fig. 5: Bits in the shared s variable.

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 Fig. 5, 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 values more readable for this presentation.) In our 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^3LP. 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 does 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 15), then checks if it is the last request of the phase to complete (Line 16). If so, $\mathcal{R}_i^{r1}$ clears the bits of s that correspond to its type (Line 17) and sets the head to be the next ticket value (Line 18). This ticket may already be held by a request in the R^3LP_LOCK procedure.

[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: procedure R3LP_Unlock(p: ptr to type_state)
14:     var ticket: unsigned int
15:     ticket := fetch&add(p→out, 1)
16:     if p→sat = ticket:                          ▷ Last request of phase to finish
17:         fetch&and(s, ∼(R1_BITS))                                  ▷ Clear R1_BITS
18:         p→head := ticket + 1                                          ▷ Update head
```

Based on the above description and implementation, we state the following lemma.

Lemma 2 *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 longest 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 17. 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 1 *The bound given in Lemma 2 is tight.*

Proof This is illustrated in Ex. 3 and Fig. 4 with $\mathcal{R}^{r1}_3$. □

Corollary 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 above, 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.

4 The Fast RW-RNLP

Our proposed 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, we present the structure of the fast RW-RNLP and provide abstract pi-blocking analysis. We then present pi-blocking analysis for the fast RW-RNLP with the R^{3}LP. Finally, we describe the RW-RNLP* and provide pi-blocking analysis for the fast RW-RNLP with the RW-RNLP*.

4.1 Protocol Structure

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 overhead; **(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. To achieve Goals (ii) and (iii), we separate nested and non-nested requests; in Sec. 4.2 and Sec. 4.5 we show these goals can be achieved with the R^{3}LP or

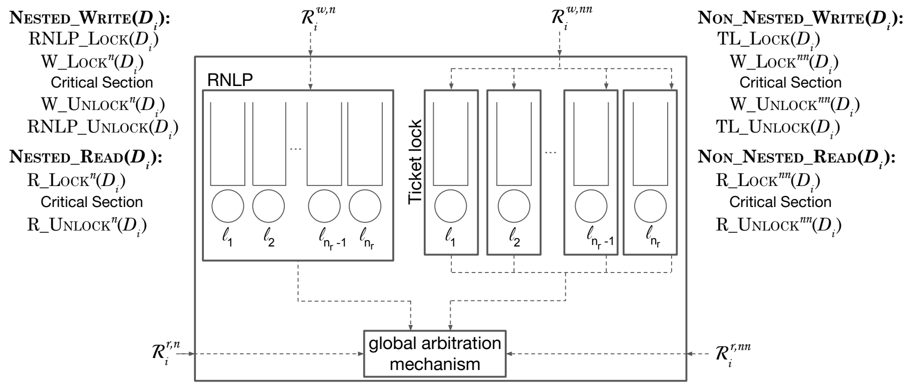

Fig. 6: Fast RW-RNLP structure.

the RW-RNLP*, respectively. (Between these two protocols, there is a trade-off between optimizing for better analytical bounds and optimizing for better runtime performance.) We address Goal (i) in Sec. 5 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 Fig. 6, we use ordinary (not phase-fair) mutex ticket locks (TLs) [38] and the RNLP [50].

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 *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 that the RNLP provides mutex sharing and supports nested requests. Under it, the worst-case pi-blocking of any request is $O(m)$ [50]. More specifically, we present the following lemma about the RNLP used in this context.

Lemma 4 *[50] 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, such as the C-RNLP, could be used to arbitrate between nested write requests without changing the overall structure of the fast RW-RNLP. We will continue to assume the RNLP is used, unless indicated otherwise.

To arbitrate between the two types of write requests, as well as read requests, a global arbitration mechanism is needed. Once a non-nested (resp.,

nested) write request is granted the resource-specific lock by a TL (resp., the RNLP), it may enter the global arbitration mechanism, as depicted in Fig. 6. 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 our 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 Fig. 6. Rule P3 is also trivially upheld in systems with only single-writer resources, which is a common use case we consider later.

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 paper, we consider two ways of implementing the global arbitration mechanism: the R^{3}LP, presented earlier in Sec. 3, and a restricted variant of the RW-RNLP, the RW-RNLP*, which we present later in Sec. 4.3. Fig. 6 shows how each type of request uses these locking protocols.[4]

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

Theorem 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) *$C_i \cdot (L^{w,nn}_{max} + G^{w,nn}) + G^{w,nn}$ 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 C_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

[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 ticket lock. Additionally, each such request must then execute its critical section for up to $L_{max}^{w,nn}$ time units. Thus, $\mathcal{R}_i^{w,nn}$ may wait up to $C_i \cdot (L_{max}^{w,nn} + G^{w,nn})$ time units before invoking the global arbitration protocol (Lemma 3), after which it may experience an acquisition delay of up to $G^{w,nn}$ time units. This yields a worst-case acquisition delay of $C_i \cdot (L_{max}^{w,nn} + G^{w,nn}) + G^{w,nn}$ time units for $\mathcal{R}_i^{w,nn}$.

A request $\mathcal{R}_i^{w,n}$ 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 4 and Rule P2). By using the same argument as before and applying Lemma 4, the worst-case acquisition delay of $\mathcal{R}^{w,n}$ is $(m-1) \cdot (L_{max}^{w,n} + G^{w,n}) + G^{w,n}$ time units. □

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

Based on the structure of the fast RW-RNLP provided in the previous section, we discuss how to apply the R^{3}LP as the global arbitration protocol. We then show that the worst-case acquisition delay a request of each type may experience under the fast RW-RNLP with the R^{3}LP achieves Goals (ii) and (iii) presented in Sec. 4.1.

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. For this purpose, we can use the R^{3}LP. The three types of requests processed by our application of the R^{3}LP are non-nested write, nested write, and read requests.

We now derive bounds on the worst-case acquisition delay that any request can experience under the fast RW-RNLP with the R^{3}LP. We distinguish between 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 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) $C_i \cdot (2L_{max}^w + L_{max}^r) + L_{max}^w + L_{max}^r$ *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) $C_i \cdot (3L^w_{max} + L^r_{max}) + 2L^w_{max} + L^r_{max}$ *time units, if* $\mathcal{R}^{w,nn}_i$ *is a non-nested write request and nested write requests may be active while* $\mathcal{R}^{w,nn}_i$ *is active;*
(v) $(m-1) \cdot (3L^w_{max} + L^r_{max}) + 2L^w_{max} + L^r_{max}$ *time units, if* $\mathcal{R}^{w,n}_i$ *is a nested write request.*

Proof Cases (ii), (iv), and (v) follow directly from Lemma 2 and Theorem 1. Here, $G^r = G^{w,nn} = G^{w,n} = 2L^w_{max} + L^r_{max}$.

When no nested write requests are active, as in cases (i) and (iii), the above statements follow from Corollary 2 and Theorem 1. Here, $G^r = G^{w,nn} = L^w_{max} + L^r_{max}$; there is no nested write phase. □

Theorem 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), we note that the worst-case pi-blocking under the RW-RNLP is $O(1)$ for read requests and $O(m)$ for write requests [52].[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 Sec. 5.

4.3 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^r_a and Q^w_a, for unsatisfied read and write requests, respectively.

Example 4 We will use Fig. 7 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. 7(a), the write request $\mathcal{R}^w_1$ 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 $\mathcal{R}^w_1$ is satisfied, it is not in any of the queues. Similarly, the read request $\mathcal{R}^r_2$ 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. The following are general rules that define how requests are processed.

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

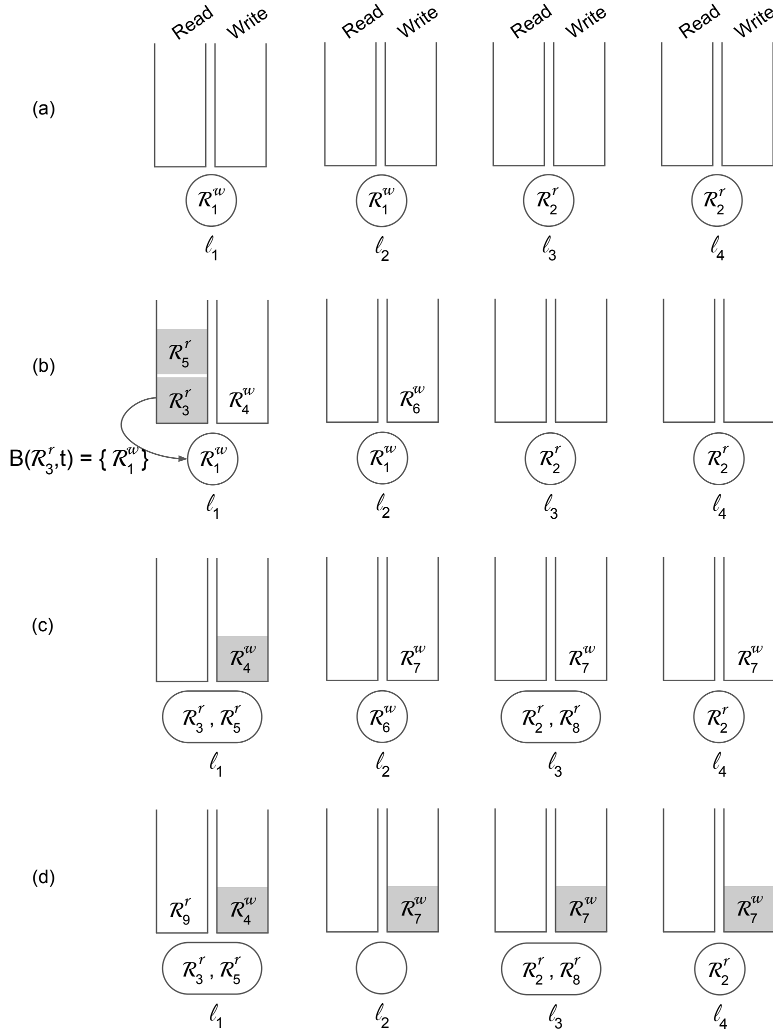

Fig. 7: Example illustrating the rules of the RW-RNLP*.

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^r_a (if it is a read request) or Q^w_a (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 4 (cont'd) Moving from inset (a) to inset (b) in Fig. 7, 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.

Fig. 7(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 in Sec. 4.6.

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 [52]). 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, we use $E(Q_a^w)$ to denote the earliest-timestamped unsatisfied write request for resource ℓ_a.

Example 4 (cont'd) In Fig. 7(b), $E(Q_2^w) = \mathcal{R}_6^w$.

Definition 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. 2).[6] (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.

Definition 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. 1),[6] and ℓ_a is not write locked. $\mathcal{R}_i^w$ remains entitled until it is satisfied.

Example 4 (cont'd) In Fig. 7(b), $\mathcal{R}_3^r$ and $\mathcal{R}_5^r$ are both entitled (Def. 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. 2). Entitled requests are indicated in Fig. 7 by gray shading.

[6] Entitlement is a property of a request, and Def. 1 and Def. 2 give conditions upon which a request becomes entitled in terms of the entitlement of other requests. Therefore, while Def. 1 and Def. 2 reference each other parenthetically to aid the reader, they are not in fact circularly defined.

In Fig. 7(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. 7(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 4 (cont'd) In Fig. 7(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. 7(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. 7, 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 4 (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. 7(b). When $\mathcal{R}_1^w$ later completed at some time t, as shown in Fig. 7(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. 7(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 4 (cont'd) When $\mathcal{R}_6^w$ was issued prior to the system state depicted in Fig. 7(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. 7(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.

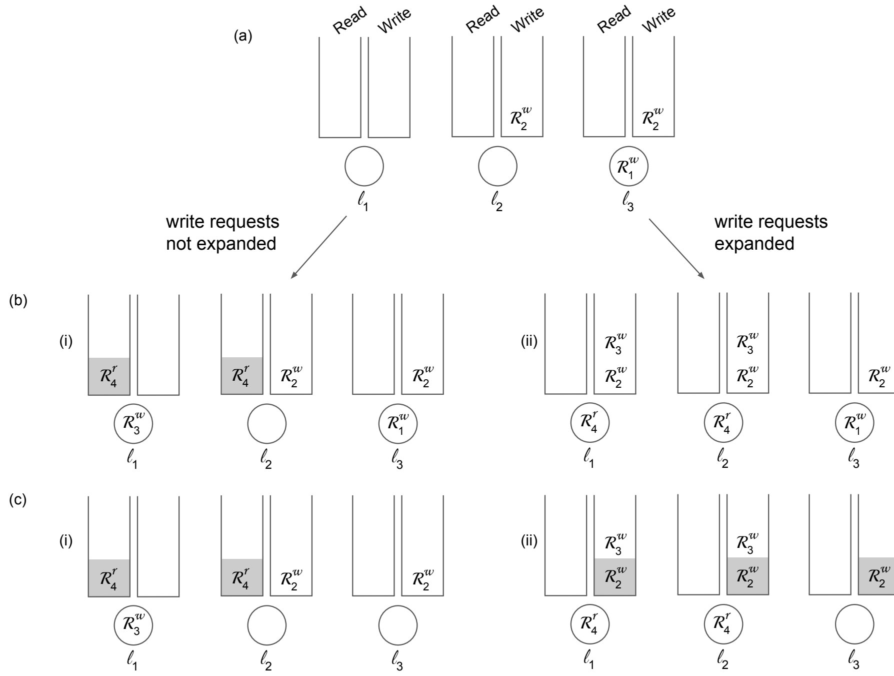

Fig. 8: System states without write expansion are labeled **(i)**, and states with write expansion (used in the RW-RNLP) are labeled **(ii).**

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 5 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. 8. 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.

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. 5 and proven in [52]. 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.

4.4 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*. The properties needed to derive acquisition-delay bounds are stated below. Lemma 5 and Theorem 3 were proved in [52] (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*. The remaining properties either require new proofs or are entirely new. We illustrate each of these properties by referring to our prior example.

Lemma 5 *Under the RW-RNLP*, a write request $\mathcal{R}_i^w$ experiences acquisition delay of at most L^r_{max} time units after becoming entitled.*

Example 4 (cont'd) In insets (c) and (d) of Fig. 7, $\mathcal{R}_4^w$ 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. 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^r_{max} 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 *Under the RW-RNLP*, the worst-case acquisition delay of a read request $\mathcal{R}_i^r$ is at most $L^w_{max} + L^r_{max}$ time units.*

Example 4 (cont'd) Consider $\mathcal{R}_9^r$ in Fig. 7(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^r_{max} time units, and then the write request $\mathcal{R}_4^w$ could take up to L^w_{max} additional time units.

Lemma 6 below is very similar to Lemma 2 in [52] 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 6 *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 5, it will become satisfied within L_{max}^r time units. Otherwise, by Def. 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. 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 Def. 2, $\mathcal{R}_i^w$ will be entitled. By Lemma 5, $\mathcal{R}_i^w$ 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 release.

Corollary 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.*

Similarly, we bound the time it takes the earliest-timestamped active write request to become satisfied in the special scenario of no active nested requests.

Lemma 7 *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 6 only in how Case (ii) in that proof is addressed. For Case (ii) in the context of Lemma 7, if the non-nested request $\mathcal{R}_x^{r,nn}$ is entitled, then by Def. 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 7. 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 5. □

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 8 *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 $2L^w_{max} + L^r_{max}$ 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 6, 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 9 *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 $4L^w_{max} + 2L^r_{max}$ 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 8, $\mathcal{R}_x^{w,n}$ will be the earliest-timestamped request for *all* of its requested resources within $2L^w_{max} + L^r_{max}$ time units. By Lemma 6, $\mathcal{R}_x^{w,n}$ 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, $\mathcal{R}_i^{w,nn}$ 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 4, given next, provides our desired acquisition-delay bounds. Together with Theorem 3, this theorem implies that all pi-blocking bounds under the RW-RNLP* are $O(1)$.

Theorem 4 *Under the RW-RNLP*, the worst-case acquisition delay of a write request $\mathcal{R}_i^w$ is:*
(i) *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) *$L^w_{max} + L^r_{max}$ 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) *$5L^w_{max} + 3L^r_{max}$ 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;*
(iv) *$3L^w_{max} + 2L^r_{max}$ time units, if $\mathcal{R}_i^{w,n}$ is a nested request.*

Proof In Case (i), by Lemma 9(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 7, it will be satisfied within L^r_{max} time units.

In Case (ii), by Lemma 9(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 6, it will be satisfied within $L^w_{max} + L^r_{max}$ time units.

In Case (iii), by Lemma 9(ii), $\mathcal{R}_i^{w,nn}$ will be the earliest-timestamped active write request for its lone requested resource within $4L^w_{max} + 2L^r_{max}$ time units. By Lemma 6, 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 (iv), by Lemma 8, $\mathcal{R}_i^{w,n}$ 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 6, 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. □

In an appendix we show that all of the blocking bounds in Theorem 4 are *tight*, *i.e.*, scenarios exist in which these exact bounds occur (see Sec. A.1). Note that, by Theorem 3 and Theorem 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 $L^w_{max} + L^r_{max}$ and L^r_{max} time units, respectively.

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

Referring back to the fast RW-RNLP structure in Fig. 6, 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 accessing 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 (we consider L_{max} 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 $O(C_i)$ 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. 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 we show in Sec. 5 that the fast RW-RNLP outperforms the RW-RNLP in practice.

Theorem 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* $\mathcal{R}_i^r$ *is a read request;*
(ii) $C_i \cdot (L^w_{max} + L^r_{max}) + L^r_{max}$ *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;*
(iii) $C_i \cdot (2L^w_{max} + L^r_{max}) + L^w_{max} + L^r_{max}$ *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;*
(iv) $C_i \cdot (6L^w_{max} + 3L^r_{max}) + 5L^w_{max} + 3L^r_{max}$ *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;*

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

Proof Each case follows directly from Theorems 1, 3, and 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 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 overhead.

4.6 RW-RNLP* Implementation

Of the building blocks used to construct the fast RW-RNLP, the TL and the RNLP have existing implementations [17,50]. In Sec. 3 we provided an implementation of the R^{3}LP. Thus, it remains for us to provide an implementation of the RW-RNLP*. Recall that we focus on the 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 [52]. 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.

Our implementation of the RW-RNLP* is based on the same ideas underlying PF-TLs. We begin by describing the shared variables we use to track requests and then present pseudocode for each type of request.

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
```

Shared variables of the RW-RNLP.* In our 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 [17]. Counters *win*

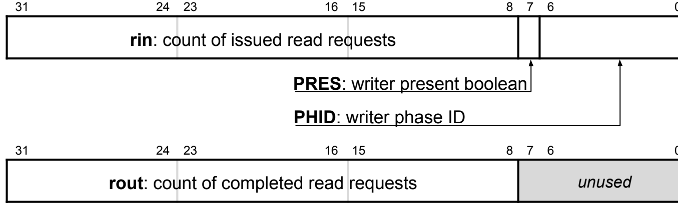

Fig. 9: Bits in the per-resource *rin* and *rout* variables. (A very similar figure appears in [17].)

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. 9. Listing 3 shows various constant bitmasks used in our 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.

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: 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, ∼(WBITS))                             ▷ Clear WBITS
16:     ℓ→wout := ℓ→wout + 1
```

Non-nested requests in the RW-RNLP.* The lock and unlock routines for non-nested requests in our implementation are shown in Listing 4. These are *nearly identical to those for the PF-TL* [17], which to our knowledge is *the most efficient reader/writer lock for single-resource requests proposed to date.* A non-nested read request $\mathcal{R}_i^{r,nn}$ 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 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 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: procedure R*_Unlock^n(D: set of ptr to res_state)
14:     for each ℓ in D:
15:         atomic_add(ℓ→rout, RINC)

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

28: procedure W*_Unlock^n(D: set of ptr to res_state)
29:     for each ℓ in D:
30:         fetch&and(ℓ→rin, ~(WBITS))                                   ▷ Clear WBITS
31:         ℓ→wout := ℓ→wout + 1
```

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] In the analysis in Sec. 4.4, we assumed that enqueueing takes no time; that is not the case in practice. We introduced this extra phase to handle a corner case in which unnecessary writer blocking occurs as a result

[7] This extra phase erroneously combined lines 3–6 into a single loop in the conference paper [40], given there as Listing 3, but the source code was correct. It has been corrected here.

of enqueueing taking some time; this corner case is explored in Sec. A.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 5.2.

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 coordinating access to the lock state with a reader/reader locking protocol (R^2LP). We define the phases of the R^2LP 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^2LP lock and unlock routines in Lines 7, 10, 21, and 25 specify their type as an input parameter. While using the R^2LP introduces blocking overhead, this overhead is only $O(1)$, as shown in [41]. This is preferable to the blocking overhead that would result from using a mutex lock to prevent race conditions.

Clearly, the routines in our 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 Sec. A.3). When viewed in this way, the routines can be shown to support the rule-based specification of the RW-RNLP* given earlier.

5 Evaluation

To evaluate both variants of our new protocol, we explored the tradeoffs between overhead and blocking via user-space experiments. We also conducted a large-scale overhead-aware schedulability study to explore the impact of the worst-case acquisition delays from Theorems 2 and 5 on system schedulability.

5.1 Overhead and Blocking

We conducted a user-space experimental evaluation of both variants of the fast RW-RNLP, for which we compared lock/unlock overhead and observed blocking times recorded under a variety of scenarios. Given the focus of this paper, we were particularly interested in overhead and blocking times for non-nested requests. We conducted our experiments on a dual-socket, 18-cores-per-socket, 2.3 GHz Intel Xeon E5-2699 platform running Ubuntu 14.04.

In our experiments, we varied a number of experimental parameters, 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). We considered the following parameter ranges: $n \in \{2, 4, ..., 36\}$, $\mathbb{D} \in \{1, 2, ..., 10\}$, $L_i \in \{0\mu s, 10\mu s, ..., , 100\mu s\}$, and nested and read probabilities independently in $\{0.0, 0.1, ..., 1.0\}$. We define a scenario 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 our graphs, we plot these worst-case values, which were obtained by computing the 99th 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). In the course of our experiments, we produced hundreds of graphs. Our protocol implementations and the full set of graphs can be found online.[8]

Overhead 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, using the experimental parameters defined above.

In designing both fast RW-RNLP variants, 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 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). A few graphs that are exemplars of trends seen generally are discussed in the following observations.

Obs. 1 *For non-nested read requests (resp., non-nested write requests), the fast RW-RNLP with the RW-RNLP* and PF-TLs exhibit comparable overhead (resp., higher overhead).*

This observation is supported by Fig. 10(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

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

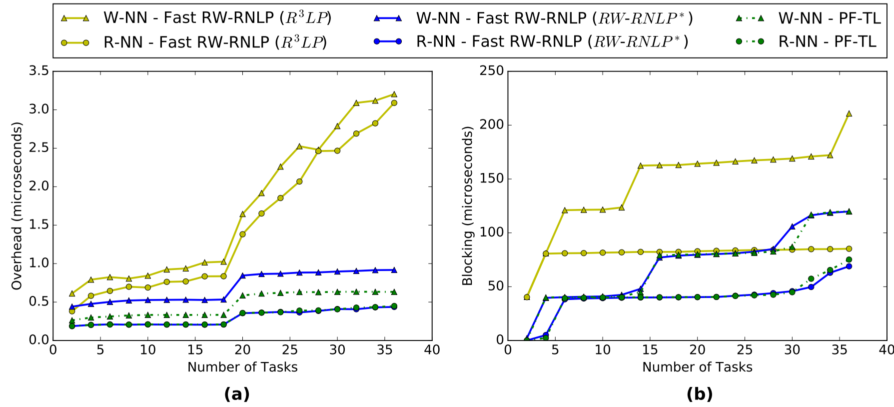

Fig. 10: **(a)** Overhead and **(b)** blocking for non-nested read and write requests when using PF-TLs versus both variants of the fast RW-RNLP. For each request $\mathcal{R}_i$, $L_i^r = 40\mu$s, $L_i^w = 40\mu$s, $n_r = 64$, $|D_i| = 1$. Requests were randomly chosen to be a read (or a write) with probability 0.5.

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 holds 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 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 increases under both protocols. This is because, beyond a task count of 18, tasks are executing on both sockets of the considered platform.

Obs. 2 *The fast RW-RNLP with the R*^{3}*LP exhibits higher overhead than PF-TLs and the fast RW-RNLP with the RW-RNLP*.*

This trend is seen in Fig. 10(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.

Obs. 3 *In general, overhead increases when using two sockets instead of one.*

This trend is seen in Fig. 10(a), discussed earlier, and also in Fig. 11(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 Fig. 10(a), overhead under the fast RW-RNLP with the RW-RNLP* is

Table 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	$C_i \cdot (2L^w + L^r) + L^w + L^r$	$C_i \cdot (L^w + L^r) + L^r$
$\mathcal{R}^{w,nn}$	3	$C_i \cdot (2L^w + L^r) + L^w + L^r$	$C_i \cdot (2L^w + L^r) + L^w + L^r$
$\mathcal{R}^{w,nn}$	1	$C_i \cdot (3L^w + L^r) + 2L^w + L^r$	$C_i \cdot (6L^w + 3L^r) + 5L^w + 3L^r$
$\mathcal{R}^{w,n}$	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 (resp., L^r) is used here to denote L^w_{max} (resp., 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 $\mathcal{R}^w$.

never more than around 1.0μs. This value is quite small compared to the 40μs critical-section length. 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.

Obs. 4 *In scenarios with only non-nested requests, the fast RW-RNLP with the RW-RNLP* and PF-TLs exhibit nearly identical blocking.*

This observation is clearly supported by Fig. 10(b). Together with Obs. 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.

Obs. 5 *In general, the fast RW-RNLP with the R^{3}LP exhibits 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. We suspect that the particular request issuance order required to generate the worst-case is not occurring during our experiments. Recall, however, that the R^{3}LP has lower analytical worst-case blocking bounds, as shown in Sec. 4.2 and 4.5 and summarized in Table 1.

Obs. 6 *In scenarios with both nested and non-nested requests, overhead for write requests tends to be much lower under both fast RW-RNLP variants than under the RW-RNLP.*

This observation is supported by Fig. 11(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 Fig. 8), 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 [52]. Note that, under the RW-RNLP, write expansion forces non-nested write requests to be processed like nested ones.

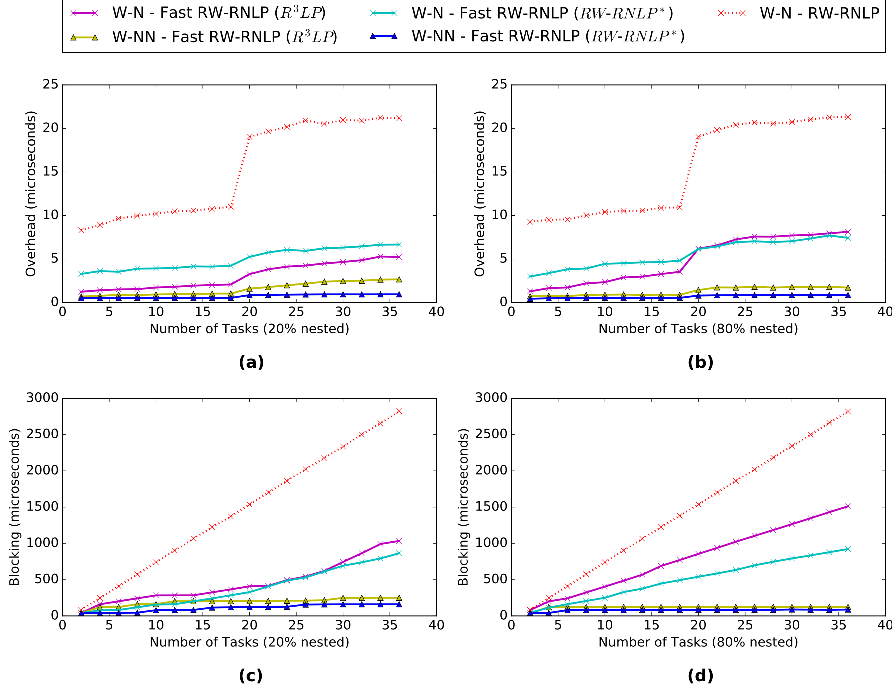

Fig. 11: **(a), (b)** Overhead and **(c), (d)** blocking for nested and non-nested write requests under the RW-RNLP and the fast RW-RNLP. Here, $L_i^r = 40\mu$s, $L_i^w = 40\mu$s, $n_r = 64$, $|D_i| = 1$, for non-nested requests, and $|D_i| = 4$, for nested requests. Requests were chosen to be a read (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 Fig. 8), D_i was inflated to include all 64 resources for writes under the RW-RNLP.

Obs. 7 *In scenarios with both nested and non-nested requests, blocking for write requests tends to be much lower under both fast RW-RNLP variants than under the RW-RNLP.*

This observation is supported by Fig. 11(c) and (d), which plot recorded worst-case blocking times associated with the scenarios in Fig. 11(a) and (b). For $m = 36$, blocking was up to 18 times lower (resp., 12 times lower) under the fast RW-RNLP with the RW-RNLP* (resp., with the R^{3}LP) than under the RW-RNLP; write expansion increases resource contention, which increases blocking times of the RW-RNLP.

Obs. 8 *Non-nested write requests exhibit contention-sensitive blocking under the fast RW-RNLP variants but not the RW-RNLP.*

This observation is also supported by Fig. 11(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;

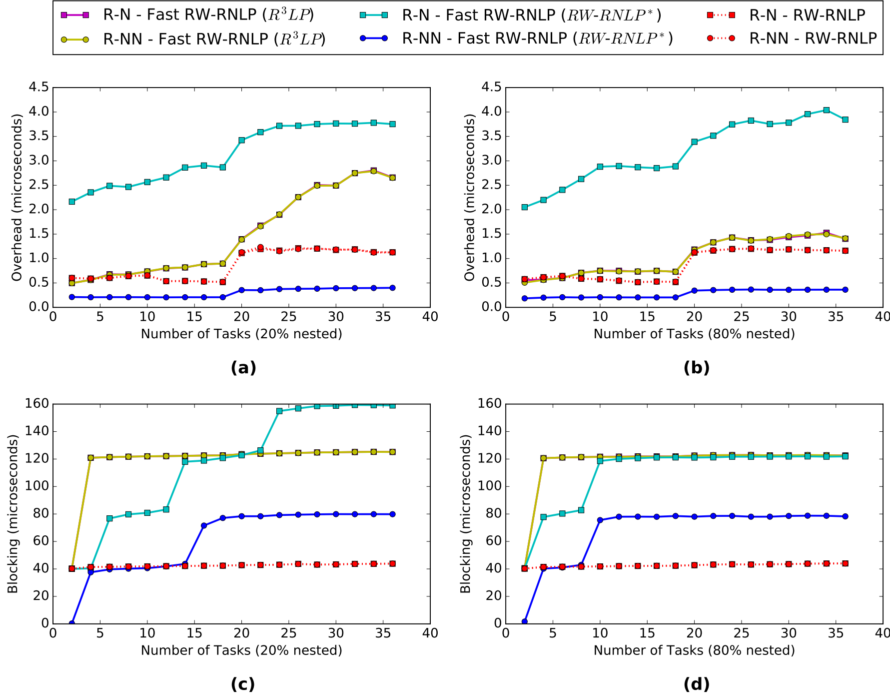

Fig. 12: **(a), (b)** Overhead and **(c), (d)** blocking for nested and non-nested read requests under the RW-RNLP and the fast RW-RNLP, in the same scenario as in Fig. 11.

with more tasks, more contention is possible, and we expect a slow linear growth of contention (and thus blocking) with the number of tasks. 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 $O(C)$, but instead $O(m)$. This translates to a faster linear growth of blocking, as in Fig. 11(c) and (d).

Notice that Fig. 11 pertains to write requests. The corresponding read request results are shown in Fig. 12. Both overhead and blocking are 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 1.

Of relevance to the analysis presented in Sec. 4, Fig. 13 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. 11(b) the number of tasks was varied, and the critical-section length was held constant; the points in Fig. 11(c) at $m = 36$ are the same as those in Fig. 13 for $L_i = 40\mu$s. Note that varying m effectively modifies the term C_i for each request $\mathcal{R}_i$.

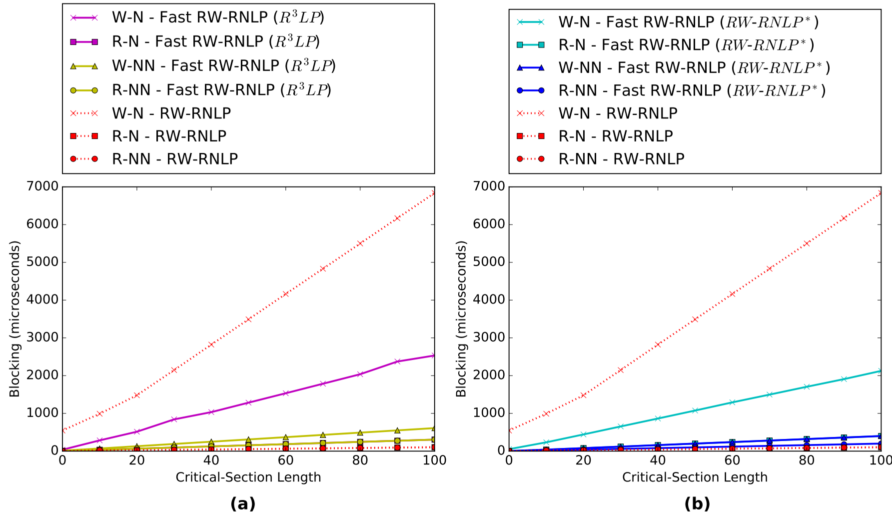

Fig. 13: Blocking for nested and non-nested write requests under the RW-RNLP and the fast RW-RNLP. The critical-section length varies, $m = 36$, $n_r = 64$, $|D_i| = 1$, for non-nested requests, and $|D_i| = 4$, for nested requests. ($|D_i|$ is inflated to 64 under the RW-RNLP as above.) A request was chosen to be a write with probability 0.5.

Obs. 9 *Blocking time scales linearly with critical-section length for both the fast RW-RNLP variants and the RW-RNLP.*

Fig. 13 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 both variants of 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 variants, 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.

5.2 Schedulability study

The results presented in Sec. 5.1 demonstrate the tradeoffs between protocols in experimentally measured overhead and blocking times. In this section, we present an evaluation of the two fast RW-RNLP variants on the basis of hard

real-time schedulability. Our large-scale study varied a range of parameters, detailed below, and took over 100 CPU-days on the platform described above.

We begin by introducing several additional constraints that can be applied to tighten the computed blocking. Then we discuss each protocol we analyze. Finally, we present the range of our schedulability study and key findings.

Constraints. For each task system, we calculated blocking using the worst-case acquisition delay bounds presented in Table 1. However, we tightened these bounds using several constraints. Instead of accounting for the system-wide worst-case critical section as repeatedly causing L_{max} blocking for other requests, we impose *period-based constraints* 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. We call this constraint on blocking the *FIFO constraint.* Similarly, the number of critical sections of read requests is limited by the number of write requests that can be counted. We refer to this 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 1), we use that number of requests for the *contention constraint.* More details and examples of these constraints can be found in Sec. A.4.

Protocols evaluated. We evaluated four protocols: 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 latter three protocols are as described above, and the PF-TL is applied to protect the group of all resources; that is, we statically group all resources and protect this group with a standard PF-TL, eliminating all nesting. We refer to this application of a PF-TL as a group PF-TL.

Experimental setup. We used SchedCAT [1], an open-source real-time schedulability test toolkit, to randomly generate task systems, implement blocking bound computations, and check for schedulability on an 18-core platform with global EDF scheduling. We varied a wide range of system parameters. For each set of parameters, we generated task sets with system utilizations in $\{2.0, 2.5, ..., 18.0\}$. We examined tasks sets with *short* ($[3, 33]$ms), *moderate* ($[10, 100]$ms), and *long* ($[50, 250]$ms) task periods. 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, we computed the impact of blocking on each request given the constraints discussed above. We did not consider most types of overhead, such as migration overhead, release blocking, and other overhead sources that impact each scheme similarly. However, we did account for the overhead incurred by using a specific locking protocol; we applied the appropriate overhead values for nested and non-nested read and write requests based on the experimental results presented in Sec. 5.1. We chose the maximum values of lock and unlock overhead from relevant scenarios (eliminating scenarios with 100% read requests and rerunning each protocol evaluation for a critical-section length of 1μs, the shortest critical section used in our 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 we applied the correct overhead measurement based on the given scenario. For all other protocols, overhead values were chosen only based on request type (*e.g.*, non-nested read).

Our schedulability experiments resulted in 960 plots. The full set is available online,[8] and we highlight a few interesting trends here.

Schedulability versus nested probability. In Fig. 14, 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 by Baruah's G-EDF schedulability test [9] after applying the appropriate blocking bounds and protocol overhead values. 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 Fig. 14, 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, we present data summarizing all results. For each of the 960 graphs, we compute the *schedulable utilization area* (SUA) of each protocol, which is the area under the curve for that protocol as approximated by a midpoint Riemann sum. In general, a higher SUA indicates better schedulability. We present a breakdown of the number of times each protocol was the best (in terms of SUA) by scenario in Table 2. We filtered out all scenarios in which all four protocols performed equally (within 2% of each other). The highest entry per nested probability is shown in bold.

Obs. 10 *For task systems in which non-nested requests are the common case, the fast RW-RNLP with the R^3LP outperforms the RW-RNLP and the group PF-TL.*

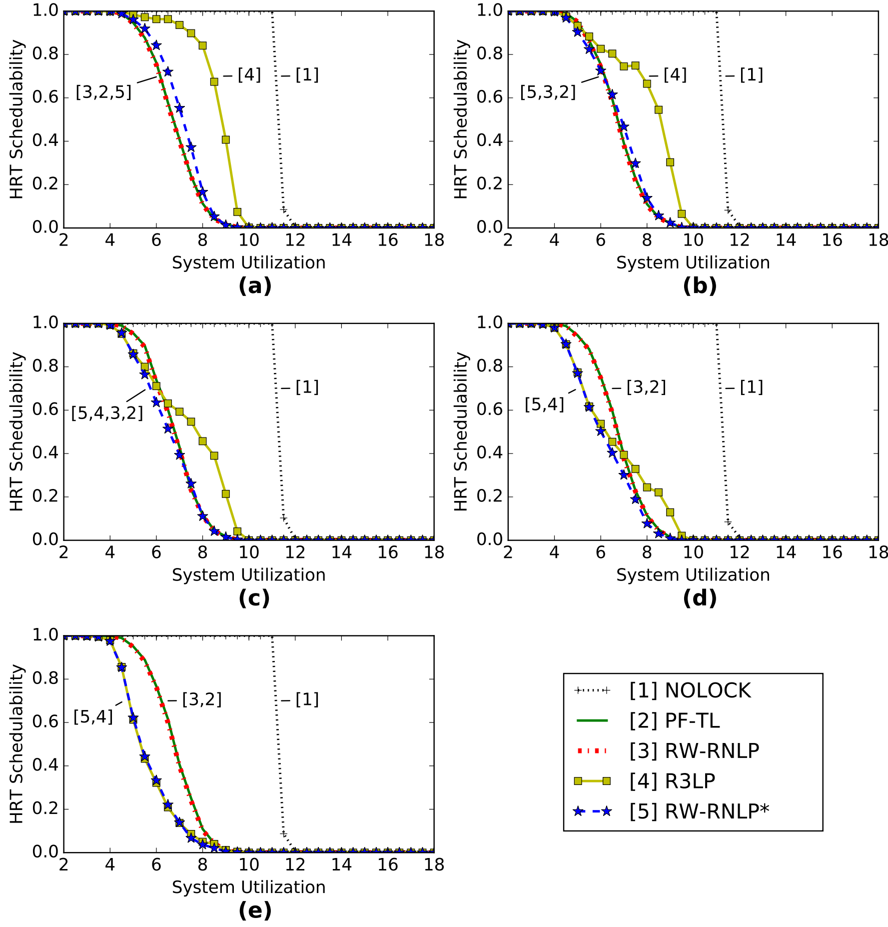

Fig. 14: Hard real-time schedulability results with varying nested probabilities for the scenario with medium task utilizations, long periods, long critical-section lengths, and read probability 0.8. Nested probabilities are **(a)** 0.01, **(b)** 0.05, **(c)** 0.1, **(d)** 0.2, and **(e)** 0.5.

Fig. 14(a), (b), and (c) reflect the trends that we observe 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^{3}LP tended to perform as well or better than the two existing protocols. This trend is highlighted in Table 2.

Obs. 11 *For most task systems we explored in which 50% of requests were nested, the group PF-TL and RW-RNLP outperform the other protocols.*

This is reflected in Fig. 14(e) and quantified in Table 2.

Schedulability versus task utilization and period. In Fig. 15, schedulability curves for each protocol are shown for task systems with short critical-section

Table 2: Best protocols per scenario by SUA

Task Util.	Nested Prob.	PF-TL	RW-RNLP	R^{3}LP	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.

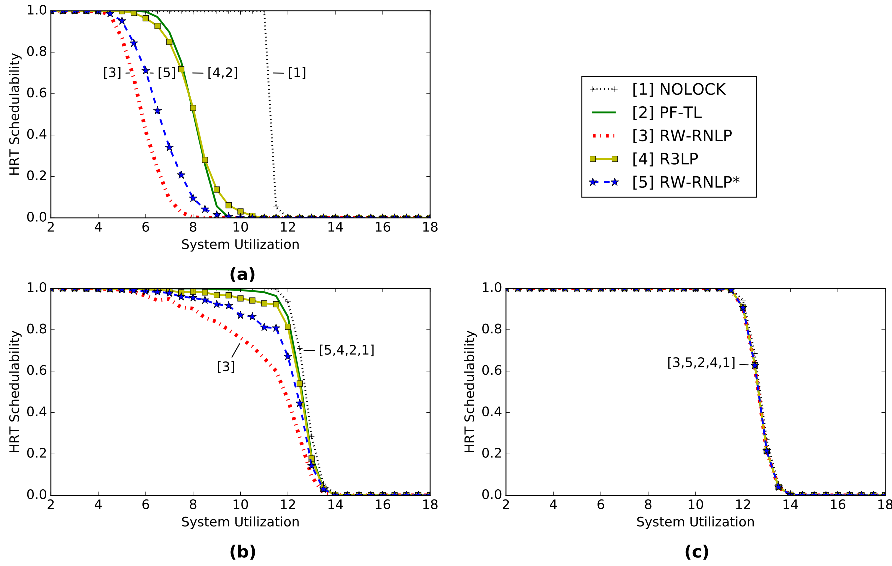

Fig. 15: Hard real-time schedulability results with varying task utilizations and periods for the scenario with short critical-section lengths, nested probability 0.1, and read probability 0.2. Task utilizations and periods are, respectively, **(a)** medium and short, **(b)** heavy and short, and **(c)** heavy and long.

lengths, nested probability 0.1, read probability 0.2, and all tasks making requests.

Obs. 12 *The fast RW-RNLP with the R^{3}LP results in higher schedulability than the fast RW-RNLP with the RW-RNLP* in most task systems.*

Table 3: Relative SUAs

Task Util.	Nested Prob.	PF-TL	RW-RNLP	R^3LP	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.

This trend can be observed in Fig. 14 and Fig. 15(a) and (b). In some task systems, the fast RW-RNLP variants display almost identical schedulability (as shown in Fig. 15(c)), and in very few task systems does the RW-RNLP* variant outperform the R^3LP variant. This trend is also reflected in Table 2, in which the R^3LP is a better choice more often than the RW-RNLP*.

For each group of scenarios, we summed the SUA of all scenarios in the group. In Table 3, we present 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, we bold the highest entry per nested probability.

Obs. 13 *Given a scenario, changing from medium task utilization to heavy task utilization tends 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 Fig. 15(a) and (b), as well as by Table 3. In Table 3, all relative SUAs increase when the task utilization changes from medium to heavy.

Obs. 14 *For some task systems, the locking protocol chosen has very 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 Fig. 15(c). In scenarios with heavy task utilization, this effect is even more pronounced, with approximately two-thirds of the scenarios having identical schedulability (Table 2).

6 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 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 eliminated the write-expansion rule of the RW-RNLP. In our experiments, this had a positive impact on blocking for all requests. We additionally demonstrated the benefit of using a fast RW-RNLP variant in scenarios in which non-nested resource accesses are the common case by conducting a large-scale schedulability study that incorporated locking protocol overhead.

The fast RW-RNLP has a modular structure that enables different variants to be applied in different contexts. For example, using the R^3LP or 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. 6 could be replaced by the C-RNLP to obtain contention-sensitive pi-blocking for nested requests (at the expense of higher overhead 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.

A Additional Details

This appendix provides additional details on several claims made in the body of the paper.

A.1 Tight Blocking Bounds for the RW-RNLP*

To show that each blocking bound proven for the RW-RNLP* is tight, we show 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 Sec. 4.4. In each example, requests are numbered in the order in which they were issued.

Lemma 5 bounds the acquisition delay that a write request can experience after becoming entitled to L^r_{max}.

Example 6 As shown in Fig. 16, 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 5.

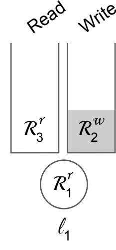

Fig. 16: A simple example that shows worst-case acquisition delay for a read request and the acquisition delay a write may experience after becoming entitled.

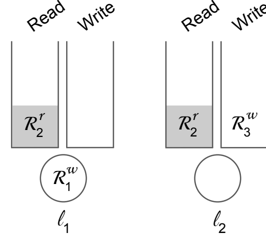

Fig. 17: An issuance order which may cause the maximum blocking after a write request $\mathcal{R}_3^w$ becomes the earliest-timestamped active write request for each of its resources, here just ℓ_2.

Theorem 3 bounds the acquisition delay a read request can experience to $L^w_{max} + L^r_{max}$.

Example 7 In Fig. 16, read request $\mathcal{R}_3^r$ experiences an acquisition delay of up to $L^w_{max} + L^r_{max}$ time units. It was issued after the issuance of requests $\mathcal{R}_1^r$ and $\mathcal{R}_2^w$, all for the same resource. $\mathcal{R}_3^r$ cannot be satisfied initially, as $\mathcal{R}_2^w$ is entitled. Therefore it waits for up to L^r_{max} 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^w_{max} time units for $\mathcal{R}_2^w$ to complete before acquiring the resource.

According to Lemma 6, a write request $\mathcal{R}_i^w$ may experience up to $L^w_{max} + L^r_{max}$ blocking after becoming the earliest-timestamped active write request for each resource in D_i.

Example 8 Similarly to the previous examples, in Fig. 17, write request $\mathcal{R}_3^w$ can experience the worst-case delay stated in Lemma 6. 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^w_{max} and L^r_{max}, respectively.

The earliest-timestamped non-nested write request with no nested requests present can experience blocking of L^r_{max}. This upper-bound proven in Lemma 7 is shown to be tight in Ex. 6 with $\mathcal{R}_2^w$ as depicted in Fig. 16.

Lemma 8 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.

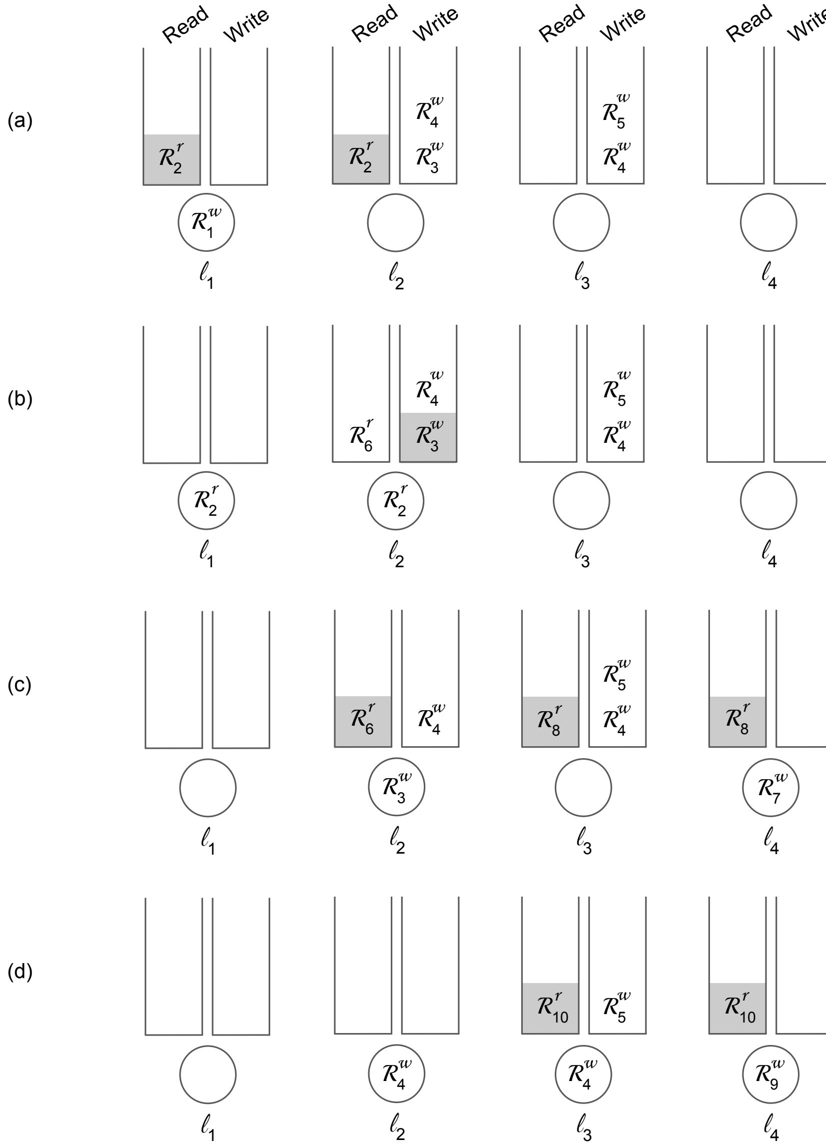

Fig. 18: A series of read and write requests that illustrate the worst-case acquisition delay for nested and non-nested write requests.

Example 9 As shown in Fig. 18(a), $\mathcal{R}_4^{w,n}$ 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 $\mathcal{R}_3^w$ has completed. Given that each of these requests could have been issued immediately after each other and that $\mathcal{R}_4^w$ will need to wait until $\mathcal{R}_1^w$, $\mathcal{R}_2^r$, and $\mathcal{R}_3^w$ complete, $\mathcal{R}_4^w$ may wait up to $2L_{max}^w + L_{max}^r$ time units to become the earliest-timestamped active write request.

Lemma 9 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 $4L^w_{max} + 2L^r_{max}$ in the presence of nested requests.

Example 10 Consider $\mathcal{R}_5^w$ in Fig. 18. This non-nested write request may wait for up to $4L^w_{max}+2L^r_{max}$ 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 Fig. 18(a), $\mathcal{R}_3^w$ may wait for up to L^w_{max} 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^r_{max} 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^w_{max} 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 $L^w_{max}+L^r_{max}$ 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^w_{max} time units, after which $\mathcal{R}_5^w$ is finally the earliest-timestamped request for D_5 after waiting for up to $4L^w_{max} + 2L^r_{max}$ time units.

Theorem 4 presents four bounds for write requests. Non-nested write requests may experience up to L^r_{max} time units of acquisition delay if no nested requests are active (illustrated in Fig. 16 and described in Ex. 6). If nested read requests may be present but no nested write requests are active, non-nested write requests may experience up to $L^w_{max} + L^r_{max}$ time units of acquisition delay, as illustrated in Fig. 17 and described in Ex. 8. The third bound presented in Theorem 4 is that non-nested write requests in the presence of nested requests may experience up $5L^w_{max} + 3L^r_{max}$ time units of acquisition delay (illustrated below). Finally, nested write requests may experience acquisition delay of up to $3L^w_{max} + 2L^r_{max}$ (also illustrated below).

Example 11 As illustrated by Fig. 18 and Ex. 10, $\mathcal{R}_5^w$ may wait for $4L^w_{max} + 2L^r_{max}$ 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 Fig. 18(d). (This is similar to the situation in Ex. 10 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 $L^w_{max} + L^r_{max}$ time units before being satisfied, making its total acquisition delay $5L^w_{max} + 3L^r_{max}$ time units.

Fig. 18 also illustrates that a nested write request, namely $\mathcal{R}_4^w$, may experience acquisition delay of $3L^w_{max} + 2L^r_{max}$. 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 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.

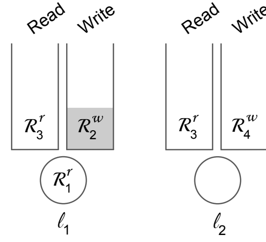

Fig. 19: Illustrates the edge case in which a write request ($\mathcal{R}_4^w$) would need to wait unnecessarily behind a nested read request ($\mathcal{R}_3^r$) if the extra code step had not been added in Listing 5.

A.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 Fig. 19. 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 13 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 released, it must wait at Line 13 (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 released, the condition at Line 13 in Listing 4 is true for its single resource ℓ_2, so it is immediately satisfied.

A.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 [32]. In the following discussion, we focus on the fast RW-RNLP with the RW-RNLP* as our example, but the fast RW-RNLP with the R^{3}LP is linearizable as well.

In the body of the paper, we claim that each routine we presented has a linearization point; this is the point at which the routine can be considered to

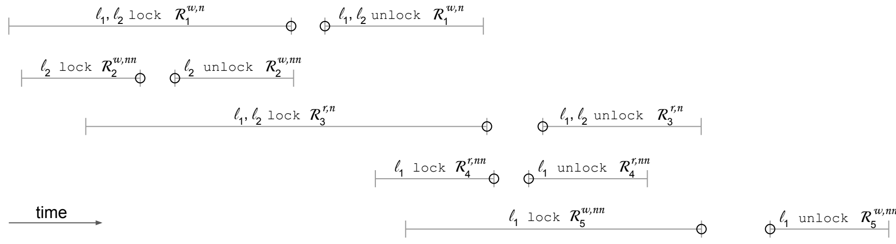

Fig. 20: Illustration of a series of lock and unlock calls by requests $\mathcal{R}_1$ through $\mathcal{R}_5$ with the linearization point of each operation shown with a circle.

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 [32].) 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 Fig. 20. 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 Fig. 20, time moves forward to the right.

Example 12 In Fig. 20, $\mathcal{R}_1^{w,n}$ 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 $D_3 = \{\ell_1, \ell_2\}$.

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 Fig. 20, 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.

A.4 Constraints used in Schedulability Study

As mentioned in the body of the paper, we tightened the calculated blocking from the the worst-case acquisition-delay bounds presented in Table 1 by applying several constraints. We illustrate these with an example using a ticket lock, and then discuss how these can be applied to more complex protocols.

Example 13 Suppose that we are working with 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, we can use the analytical bound for a ticket lock of $(m-1)L_{max}^w$, where $L_{max}^w = 60\mu$s. Then the worst-case 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, we can compute how many jobs (and thus requests) of that task may be active while a given request is active. Then we can tighten our analysis to select only the highest instances of critical sections that may delay a given request.

Example 13 (cont'd) 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. We can tighten our analysis to select the top $m-1$ instances of critical sections that may delay $\mathcal{R}_1^w$. Our new bound on the worst-case blocking after applying period-based constraints is $60+60+30+30+20=200\mu$s. Note that we do not count $\mathcal{R}_1^w$ 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 13 (cont'd) By imposing FIFO constraints, we can bound worst-case blocking to $60+30+20=110\mu$s.

Example 14 We now modify the task system presented in Ex. 13 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. We switch from using a ticket lock to using a PF-TL. When considering the blocking $\mathcal{R}_1^w$ may experience, the above analysis still holds for the write requests. Now we must account for the blocking $\mathcal{R}_1^w$ may experience due to read requests.

Note that when we consider 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 we have discussed 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, we can constrain the total number of read requests that must be included by period-based constraints and a new constraint.

Read-write constraints. Each protocol examined in this paper 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 14 (cont'd) 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. Thus, we are limited to the four highest read critical-section instances. Because our read request 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.

While we have focused on the blocking a write request may experience in the above examples, 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 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, we use the number of contending non-nested write requests for the value of contention.

Constraints applied to protocols. Recall that we evaluated four protocols: 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 $(m-1)(L^w_{max} + L^r_{max})$ 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^{3}LP, when we account for blocking in the fast RW-RNLP with the R^{3}LP, we add the highest critical-section length instances of non-nested write requests and nested write requests separately to get a tighter bound.

Based on the number of requests being generated in each scenario, we expect 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 [55].

References

1. SchedCAT: Schedulability test collection and toolkit. `http://www.mpi-sws.org/bbb/projects/schedcat`. Accessed: 2018-05-31
2. Afshar, S., Behnam, M., Bril, R., Nolte, T.: Flexible spin-lock model for resource sharing in multiprocessor real-time systems. In: Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems, pp. 41–51. IEEE (2014)
3. Afshar, S., Behnam, M., Bril, R., Nolte, T.: 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, pp. 1–11. IEEE (2017)
4. Afshar, S., Behnam, M., Bril, R., Nolte, T.: Per processor spin-based protocols for multiprocessor real-time systems. Leibniz Transactions on Embedded Systems **4**(2), 1–30 (2018)
5. Afshar, S., Behnam, M., Nolte, T.: Integrating independently developed real-time applications on a shared multi-core architecture. ACM SIGBED Review **10**(3), 49–56 (2013)
6. Afshar, S., Khalilzad, N., Nemati, F., Nolte, T.: Resource sharing among prioritized real-time applications on multiprocessors. ACM SIGBED Review **12**(1), 46–55 (2015)
7. Bacon, D., Konuru, R., Murthy, C., Serrano, M.: Thin locks: Featherweight synchronization for Java. In: ACM SIGPLAN Notices, vol. 33, pp. 258–268. ACM (1998)
8. Baker, T.P.: Stack-based scheduling of realtime processes. Real-Time Systems **3**(1), 67–99 (1991)
9. Baruah, S.: Techniques for multiprocessor global schedulability analysis. In: Proceedings of the 28th IEEE Real-Time Systems Symposium, pp. 119–128. IEEE (2007)
10. Biondi, A., Brandenburg, B., Wieder, A.: A blocking bound for nested FIFO spin locks. In: Proceedings of the 37th IEEE Real-Time Systems Symposium, pp. 291–302. IEEE (2016)
11. Block, A., Leontyev, H., Brandenburg, B., Anderson, J.: 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, pp. 71–80. IEEE (2007)
12. Brandenburg, B.: Scheduling and locking in multiprocessor real-time operating systems. Ph.D. thesis, University of North Carolina, Chapel Hill, NC (2011)
13. Brandenburg, B.: The FMLP+: An asymptotically optimal real-time locking protocol for suspension-aware analysis. In: Proceedings of the 26th Euromicro Conference on Real-Time Systems, pp. 61–71. IEEE (2014)
14. Brandenburg, B., Anderson, J.: Feather-trace: A lightweight event tracing toolkit. In: Proceedings of the 3rd International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, pp. 19–28 (2007)
15. Brandenburg, B., Anderson, J.: A comparison of the M-PCP, D-PCP, and FMLP on LITMUS^{RT}. In: Proceedings of the 12th International Conference on Principles of Distributed Systems, pp. 105–124. Springer (2008)
16. Brandenburg, B., Anderson, J.: An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS^{RT}. In: Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp. 185–194. IEEE (2008)
17. Brandenburg, B., Anderson, J.: Spin-based reader-writer synchronization for multiprocessor real-time systems. Real-Time Systems **46**(1), 25–87 (2010)
18. Brandenburg, B., Anderson, J.: 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, pp. 69–78. ACM (2011)

19. Brandenburg, B., Anderson, J.: The OMLP family of optimal multiprocessor real-time locking protocols. Design Automation for Embedded Systems **17**(2), 277–342 (2013)
20. Burns, A., Wellings, A.: A schedulability compatible multiprocessor resource sharing protocol - MrsP. In: Proceedings of the 25th Euromicro Conference on Real-Time Systems, pp. 282–291. IEEE (2013)
21. Chang, Y., Davis, R., Wellings, A.: Reducing queue lock pessimism in multiprocessor schedulability analysis. In: Proceedings of the 18th International Conference on Real-Time and Network Systems, pp. 99–108 (2010)
22. Chen, C., Tripathi, S.: Multiprocessor priority ceiling based protocols. Tech. Rep. CS-TR-3252, University of Maryland (1994)
23. Courtois, P., Heymans, F., Parnas, D.: Concurrent control with readers and writers. Communications of the ACM **14**(10), 667–668 (1971)
24. Dijkstra, E.: Two starvation free solutions to a general exclusion problem. EWD 625, Plataanstraat 5, 5671 Al Nuenen, The Netherlands
25. Elliott, G., Anderson, J.: An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems. Real-Time Systems **49**(2), 140–170 (2013)
26. Faggioli, D., Lipari, G., Cucinotta, T.: The multiprocessor bandwidth inheritance protocol. In: Proceedings of the 22nd Euromicro Conference on Real-Time Systems, pp. 90–99. IEEE (2010)
27. Faggioli, D., Lipari, G., Cucinotta, T.: Analysis and implementation of the multiprocessor bandwidth inheritance protocol. Real-Time Systems **48**(6), 789–825 (2012)
28. Gai, P., Di Natale, M., Lipari, G., Ferrari, A., Gabellini, C., Marceca, P.: A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform. In: Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 189–198. IEEE (2003)
29. Gai, P., Lipari, G., Di Natale, M.: Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In: Proceedings of the 22nd IEEE Real-Time Systems Symposium, pp. 73–83. IEEE (2001)
30. Garrido, J., Zhao, S., Burns, A., Wellings, A.: Supporting nested resources in MrsP. In: Proceedings of the Ada-Europe International Conference on Reliable Software Technologies, pp. 73–86. Springer (2017)
31. Havender, J.: Avoiding deadlock in multitasking systems. IBM systems journal **7**(2), 74–84 (1968)
32. Herlihy, M., Wing, J.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems **12**(3), 463–492 (1990)
33. Huang, H., Pillai, P., Shin, K.: 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, pp. 303–316. USENIX Association (2002)
34. Jarrett, C., Ward, B., Anderson, J.: 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, pp. 3–12. ACM (2015)
35. Joung, Y.: Asynchronous group mutual exclusion. Distributed Computing **13**(4), 189–206 (2000)
36. Keane, P., Moir, M.: A simple local-spin group mutual exclusion algorithm. In: Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing, pp. 23–32. ACM (1999)
37. Lakshmanan, K., Niz, D., Rajkumar, R.: Coordinated task scheduling, allocation and synchronization on multiprocessors. In: Proceedings of the 30th IEEE Real-Time Systems Symposium, pp. 469–478. IEEE (2009)
38. Mellor-Crummey, J., Scott, M.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems **9**(1), 21–65 (1991)
39. Nemati, F., Behnam, M., Nolte, T.: Independently-developed real-time systems on multi-cores with shared resources. In: Proceedings of the 23rd Euromicro Conference on Real-Time Systems, pp. 251–261. IEEE (2011)
40. Nemitz, C., Amert, T., Anderson, J.: Real-time multiprocessor locks with nesting: Optimizing the common case. In: Proceedings of the 25th International Conference on Real-Time Networks and Systems, pp. 38–47. ACM (2017)

41. Nemitz, C., Amert, T., Anderson, J.: 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, pp. 25:1–25:24. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2018)
42. Rajkumar, R.: Real-time synchronization protocols for shared memory multiprocessors. In: Proceedings of the 10th International Conference on Distributed Computing Systems, pp. 116–123. IEEE (1990)
43. Rajkumar, R.: Synchronization in Real-Time Systems: A Priority Inheritance Approach. Kluwer Academic Publishers, Boston (1991)
44. Rajkumar, R., Sha, L., Lehoczky, J.: Real-time synchronization protocols for multiprocessors. In: Proceedings of the 9th IEEE Real-Time Systems Symposium, pp. 259–269. IEEE (1988)
45. Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority inheritance protocols: An approach to real-time synchronization. IEEE Transactions on computers **39**(9), 1175–1185 (1990)
46. Takada, H., Sakamura, K.: Real-time scalability of nested spin locks. In: Proceedings of the 2nd IEEE International Workshop on Real-Time Computing Systems and Applications, pp. 160–167. IEEE (1995)
47. Wang, C., Takada, H., Sakamura, K.: Priority inheritance spin locks for multiprocessor real-time systems. In: Proceedings of the 2nd IEEE International Symposium on Parallel Architectures, Algorithms, and Networks, pp. 70–76. IEEE (1996)
48. Ward, B.: Relaxing resource-sharing constraints for improved hardware management and schedulability. In: Proceedings of the 36th IEEE Real-Time Systems Symposium, pp. 153–164. IEEE (2015)
49. Ward, B.: Sharing non-processor resources in multiprocessor real-time systems. Ph.D. thesis, University of North Carolina, Chapel Hill, NC (2016)
50. Ward, B., Anderson, J.: Supporting nested locking in multiprocessor real-time systems. In: Proceedings of the 23rd Euromicro Conference on Real-Time Systems, pp. 223–232. IEEE (2012)
51. Ward, B., Anderson, J.: Fine-grained multiprocessor real-time locking with improved blocking. In: Proceedings of the 21st International Conference on Real-Time Networks and Systems, pp. 67–76. ACM (2013)
52. Ward, B., Anderson, J.: Multi-resource real-time reader/writer locks for multiprocessors. In: Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium, pp. 177–186. IEEE (2014)
53. Ward, B., Elliott, G., Anderson, J.: 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, pp. 280–289. IEEE (2012)
54. Wieder, A., Brandenburg, B.: 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, pp. 45–56. IEEE (2013)
55. Wieder, A., Brandenburg, B.: On the complexity of worst-case blocking analysis of nested critical sections. In: Proceedings of the 35th IEEE Real-Time Systems Symposium, pp. 106–117. IEEE (2014)
56. Yang, M., Wieder, A., Brandenburg, B.: Global real-time semaphore protocols: A survey, unified analysis, and comparison. In: Proceedings of the 36th IEEE Real-Time Systems Symposium, pp. 1–12. IEEE (2015)
57. Zhao, S., Garrido, J., Burns, A., Wellings, A.: New schedulability analysis for MrsP. In: Proceedings of the 23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp. 1–10. IEEE (2017)