Multi-Resource Real-Time Reader/Writer Locks for Multiprocessors [*]

Bryan C. Ward and James H. Anderson

Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

A fine-grained locking protocol *permits multiple locks to be held simultaneously. In the case of real-time multiprocessor systems, prior work on such protocols has considered only mutex constraints. This unacceptably limits concurrency in systems in which some resource accesses are read-only. To remedy this situation, a variant of a recently-proposed fine-grained protocol called the* real-time nested locking protocol (RNLP) *is presented that enables concurrent reads. Like the original RNLP, this reader/writer version is a "pluggable" protocol that has different variants for different schedulers and analysis assumptions. Some of these variants are asymptotically optimal with respect to priority-inversion blocking. This paper is the first to address the algorithmic and analytical intricacies that arise when attempting to support read-only resource accesses under fine-grained locking in multiprocessor real-time systems.*

1 Introduction

To exploit the performance benefits of multicore machines, which are becoming ever more ubiquitous, applications must be efficiently parallelized. This requires (among other things) efficient techniques for synchronizing accesses to shared resources by different tasks. For such techniques to be deemed "efficient," they should permit synchronizing tasks to execute concurrently where possible, as tasks executing sequentially do not benefit from the availability of multiple processors. Moreover, concurrent accesses of multiple resources by the same task should be supported, as such functionality is widely employed in practice.

When locks are used to support resource sharing, two principal approaches exist: coarse-grained and fine-grained locking. Under *coarse-grained locking*, all resources that may be accessed concurrently via operations that conflict are grouped into a single lockable entity, and a single-resource locking protocol is used. This approach, also known as *group locking* [3], is clearly detrimental to concurrency. In contrast, under *fine-grained locking*, different resources are locked individually. This enables concurrent accesses of separate resources, but issues such as deadlock become problematic.

*Work supported by NSF grants CNS 1016954, CNS 1115284, and CNS 1239135; ARO grant W911NF-09-1-0535; AFOSR grant FA9550-09-1-0549; and AFRL grant FA8750-11-1-0033. The first author was supported by an NSF graduate research fellowship.

Perhaps because of such issues, the first fine-grained locking protocol for multiprocessor real-time systems was proposed only recently, in the form of the *real-time nested locking protocol* (*RNLP*) of Ward and Anderson [19]. The RNLP is actually a "pluggable" protocol that has different variants for different schedulers and analysis assumptions. Most of these variants are asymptotically optimal with respect to worst-case priority-inversion blocking, or *pi-blocking* (see Sec. 2). Unfortunately, from the perspective of enabling concurrency, the RNLP has a serious shortcoming: it treats all resources as mutex resources that can be accessed by only one task at a time. This unacceptably limits concurrency if some accesses are read-only.

Contributions. To address this shortcoming, we present a reader/writer variant of the RNLP (the R/W RNLP for short) that allows read-only accesses to execute concurrently. The design of the R/W RNLP breaks new ground in several ways. For example, it is the first fine-grained multiprocessor real-time locking protocol that allows tasks to hold read locks and write locks simultaneously on different resources, and the first to allow read locks to be upgraded to write locks. As in the work on the original RNLP, we judge the efficacy of a locking protocol in terms worst-case pi-blocking and present protocol variants for various implementation and analysis assumptions. Some of these variants are asymptotically optimal in terms of worst-case pi-blocking. To make the R/W RNLP easier to understand, we initially assume that tasks block by *spinning* (busy waiting). We subsequently discuss variants in which blocked tasks instead *suspend* by considering how the original protocol must be changed if spinning is replaced by suspending.

Further motivation. In conducting this work, we were partially motivated by the desire to provide a foundation for supporting lock-based *transactional memory* (*TM*) in real-time multiprocessor systems. TM provides infrastructure that lifts the burden of dealing with synchronization issues from programmers in the case of memory-resident shared objects. When TM is used, code that accesses shared objects is specified as a sequential *transaction*. The underlying TM framework is responsible for properly synchronizing conflicting transactions that access common objects. The use of TM in real-time applications could greatly ease certification difficulties, provided the employed TM was designed with real-time predictability in mind.

While TM can be supported in hardware [13, 16], our interest is in *software TM* (*STM*) [18], as it can be applied to systems without hardware support. In prior work on STM, both throughput-oriented systems (the literature here is quite vast—please see [10] and the references therein) and real-time systems [1, 11, 12, 14, 15] have been considered.

In all work (known to us) on STM for real-time systems, the focus has been on non-blocking approaches. In a *non-blocking* approach, transactions may execute concurrently, at the expense of potentially needing to abort and retry conflicting transactions. In a real-time system, transactions retries must be analytically bounded for schedulability analysis. While reasonable bounds are attainable on uniprocessors [2], such bounds are difficult to obtain on multiprocessors. Moreover, making transactions abortable (so that they can be retried) usually entails making copies of all or part of an object's state. In the worst case (which is of relevance to real-time systems), this copying overhead can be expensive. Because of these issues, non-blocking mechanisms have fared poorly in comparison to lock-based ones in prior experimental studies that focus on multiprocessor schedulability [9]. Based on this evidence, we believe that STM for real-time systems should be *lock-based*.

To be practically viable, STM must properly support read-only object accesses (this is an issue that has been extensively investigated in the literature on STM [18]). In the case of lock-based real-time STM, this means that fine-grained locking techniques that can properly deal with read-only accesses are crucial. Such techniques are presented in this paper in the form of the R/W RNLP. Despite our interest in STM, the R/W RNLP is not limited to controlling memory-resident shared objects. For this reason, we use the more general term "resource" instead of "shared object" in describing our results.

Scope. Given the numerous algorithmic, systems, and language challenges that must be addressed, the development of a complete real-time lock-based STM framework is an undertaking beyond the scope of any single paper. The focus of this work is to establish the *algorithmic foundation* that is required to implement the core of STM: the transaction manager that *predictably* and *efficiently* coordinates concurrent read and write accesses. In a lock-based STM, this inherently requires a fined-grained R/W locking protocol—to this end, this paper presents the design and analysis of the R/W RNLP, which will serve as the foundation for future empirical and systems-oriented work.

Organization. In the rest of the paper, we present needed background (Sec. 2), present the R/W RNLP, its blocking analysis, and ways to add additional functionality (Sec. 3), and conclude (Sec. 4).

2 Background

We consider a system with m processors and n sporadic tasks $T_1, \ldots, T_n$. Each task T_i releases a sequence of jobs. We denote an arbitrary job of T_i as J_i. Jobs of T_i are released with a *minimum separation* of p_i time units. Each such job executes for an *execution requirement* of at most e_i time units and should complete before a specified *relative deadline* d_i time units after its release. We consider time to be continuous. A job is said to be *pending* after being released until it completes execution.

Resource model. We consider a system with q shared resources (excluding processors), $\ell_1, \ldots, \ell_q$, such as shared memory objects. When a job requires access to one or more resources, it issues a *request* to a *locking protocol*. (Note that multiple resources may be included in one request.) We denote J_i's k^{th} resource request as $\mathcal{R}_{i,k}$. A request is said to be *satisfied* when access is granted to all requested resources and *completed* when the job *releases* all requested resources. A satisfied request is said to *hold* its requested resources. The time between a request being issued and being satisfied is *acquisition delay*. The time between the satisfaction of a request and its completion is a *critical section*. If a job must wait for a resource, it can do so by either spinning or suspending. A pending job that is not suspended is *ready*.

Each resource indicated in a request is requested for either *reading* or *writing*. We say that a resource is *read* (*write*) *locked* if it is held by a request that reads (writes) it. We assume that each resource ℓ_a is subject to a *reader/writer sharing constraint*: writes of ℓ_a are mutually exclusive, but arbitrarily many reads of ℓ_a can be executed concurrently. Such a read is not allowed to modify ℓ_a. Two requests *conflict* if they include a common resource that is written by at least one of them.

Scheduling. Without loss of generality, we consider clustered-scheduled systems, in which the m processors are grouped into m/c clusters, each of size c. Tasks are statically assigned to clusters, and within each cluster, jobs are scheduled from a single ready queue. Therefore, a task can migrate among the processors within its cluster. Note that partitioned and global scheduling are special cases of clustered scheduling, in which $c = 1$ and $c = m$, respectively.

We assume a job-level fixed priority scheduling algorithm, in which each job has a constant *base priority*, but different jobs of the same task may have differing base priorities. In the locking protocols we develop, resource-holding jobs may have their base priority elevated to a higher *effective priority*, which the scheduling algorithm uses to schedule the job. Elevating the effective priority of a job is often necessary to ensure resource-holding jobs are scheduled. This can be accomplished through one of a number of mechanisms, such as non-preemptivity or priority donation [6], as will be discussed in more detail later.

Blocking. We evaluate the blocking of the presented locking protocols on the basis of their worst-case *priority-inversion blocking* (*pi-blocking*) [5]. We give here definitions concerning blocking for the case in which waiting is realized by spinning; alternate definitions for suspension-based waiting will be given later.

Def. 1. A job J_i incurs *pi-blocking* at time t if J_i is ready but not scheduled and fewer than c higher-priority jobs are ready in T_i's cluster.

For example, if a high-priority job J_h is released, but a low-priority job executing non-preemptively is preventing

J_h from being scheduled, then J_h is pi-blocked. A job may also be blocked while waiting for a resource:

Def. 2. A job J_i incurs *s-blocking* at time t if J_i is spinning (and thus scheduled) waiting for a resource.

For example, if J_i is spinning while waiting for ℓ_a, which is held by J_k, then J_i is s-blocked.

Analysis assumptions. We employ analysis assumptions similar to [4, 5, 6, 19] with respect to all claims of optimality. Specifically, for asymptotic analysis, we assume the number of processors, m, and the number of tasks, n, to be variables, and all other parameters to be constants. Examples of such constants include critical section lengths as well as the number of critical sections per job. Additionally, we assume that locking protocol invocations take zero time and all other overheads are negligible (such overheads are orthogonal to the algorithmic issue at hand and can be factored into the final analysis [4, Chaps. 3,7]).

3 R/W RNLP

The aim of this paper is to extend the original mutex RNLP [19] to enable fine-grained access to resources subject to a reader/writer (R/W) sharing constraint. Specifically, our goal is to enable non-conflicting requests to be satisfied concurrently, to the extent possible. We also desire the following additional properties.

- **R/W mixing**. A job may read some resources while writing others in a single critical section. This allows for increased concurrency for those resources that are only read.
- **R-to-W upgrading.** A job that has acquired a resource for reading may *upgrade* its read to a write. For example, a job may read a resource, and based upon the value read, decide that it needs to write that resource.
- **Incremental locking.** The resources accessed by a job within a single critical section may be requested via a sequence of requests. For example, a job may request ℓ_a, read its value, and then execute some conditional code that requests ℓ_b.

We call the protocol we obtain the *R/W RNLP* . In describing the R/W RNLP, we initially assume for simplicity that the properties above are not supported, that is:

Assumption 1. *All resources accessed within a single critical section are requested via a single request, these resources are either all read or all written, and no read request may be upgraded.*

We later explain how to support R/W mixing in Sec. 3.5, R-to-W upgrading in Sec. 3.6, and incremental locking in Sec. 3.7. Until we get to these later subsections (*i.e.*, while Assumption 1 is in place), we use the following notation. We denote the set of resources that are needed in $\mathcal{R}_{i,k}$'s critical section as $\mathcal{N}_{i,k}$. By Assumption 1, each request can be categorized as either a *read request* or a *write request*, and each critical section as either a *read critical section* or a *write critical section*. For notational clarity, we often annotate read (write) requests as $\mathcal{R}^w_{i,k}$ ($\mathcal{R}^r_{i,k}$). We denote the longest read (write) critical section length as L^r_{max} (L^w_{max}), and we let $L_{max} = \max(L^r_{max}, L^w_{max})$.

In recent work, Brandenburg and Anderson [6, 7] developed *phase-fair R/W locks*, which are asymptotically optimal for a single resource (or coarse-grained locking). Intuitively, phase-fair locks function by alternating read phases, in which all issued read requests are satisfied, and write phases, in which a single write request is satisfied. In this sense, reads concede to writes, and writes concede to reads. This results in worst-case reader blocking of $O(1)$ and worst-case writer blocking of $O(m)$. In extending the mutex RNLP to enable fine-grained R/W sharing and the properties listed above, we utilize this phasing concept.

Devising such an extension presents several challenges. Perhaps the biggest challenge is the problem of *inconsistent phases*. If both a read and a write request are waiting for two resources, one of which is read locked and the other write locked, which request should be satisfied next? As we desire optimality, any solution to this problem should ensure $O(1)$ blocking for read requests.

Having motivated some of the properties we desire of our protocol and challenges we face, we now describe the spin-based variant of the R/W RNLP, given Assumption 1.

3.1 R/W RNLP Architecture

Like the original mutex RNLP, the R/W RNLP is composed of two components, a *request satisfaction mechanism* (*RSM*) and a *progress mechanism*. The RSM orders the satisfaction of resource requests. To ensure a bounded duration of blocking, the progress mechanism may elevate the effective priority of a resource-holding job to ensure it is scheduled. The choice of progress mechanism is dependent upon how waiting is realized (spinning or suspending). When the RSM is paired with an appropriate progress mechanism, the resulting locking protocol optimally supports fine-grained resource sharing under many different analysis and implementation assumptions.

Before describing the RSM, we abstractly characterize the progress mechanism by stating two needed properties.

P1 A resource-holding job is always scheduled.

P2 At most m jobs may have incomplete resource requests at any time, at most c from each cluster.

Non-preemptive spinning fulfills these requirements:

S1 A job with an incomplete request executes non-preemptively (both while spinning and within its critical section).

From this rule we have the following lemma.

Lemma 1. *Rule S1 implies Properties P1 and P2.*

For ease of exposition, we assume this simple progress mechanism for now. Later, we specify appropriate progress mechanisms for non-spin-based implementations.

3.2 RSM

In the RSM, two queues are used per resource ℓ_a, a queue for readers, $RQ(\ell_a)$, and a queue for writers, $WQ(\ell_a)$. This

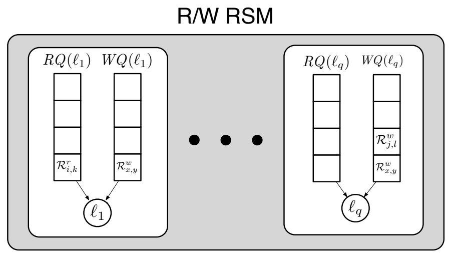

R/W RSM
$RQ(\ell_1)$ $WQ(\ell_1)$
$\mathcal{R}^r_{i,k}$ $\mathcal{R}^w_{x,y}$
ℓ_1
$RQ(\ell_q)$ $WQ(\ell_q)$
$\mathcal{R}^w_{j,l}$ $\mathcal{R}^w_{x,y}$
ℓ_q

Figure 1: Illustration of the queue structure in the R/W RSM. For each resource $\ell_a \in \{\ell_1, \ldots, \ell_q\}$, there is a read queue $RQ(\ell_a)$ and a write queue $WQ(\ell_a)$.

is depicted in Fig. 1. We assume that each read (write) request is enqueued atomically in the read (write) queue of each resource it requests. The timestamp of the issuance of each request is recorded and denoted $ts(\mathcal{R}_{i,k})$. All writer queues are order by these timestamps, resulting in FIFO queueing. We denote the earliest timestamped incomplete write request for ℓ_a (*i.e.*, the head of the $WQ(\ell_a)$) as $E(WQ(\ell_a))$. Similar to phase-fair locks [7], the queue from which requests are satisfied ($RQ(\ell_a)$ or $WQ(\ell_a)$) alternates. The techniques that govern such alternation, however, are quite different from traditional phase-fair locks because we must avoid the problem of inconsistent phases.

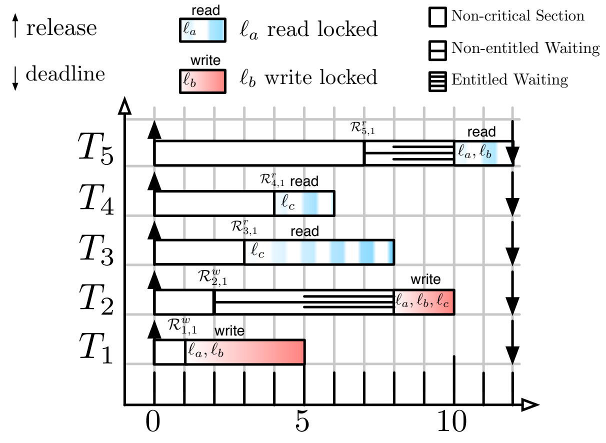
Example. As we explain the rules of the RSM, we will reference relevant parts of the example schedule in Fig. 2, which will later be explained in its entirety. In this running example, there are five tasks and a processor for each task, such that all pending jobs are scheduled. Additionally, these tasks share three resources, ℓ_a, ℓ_b and ℓ_b. At time $t = 2$, when $\mathcal{R}^w_{2,1}$ is issued, $ts(\mathcal{R}^w_{2,1}) = 2$ is established. Also, since $\mathcal{R}^w_{2,1}$ requires all three resources and since it is the only write request waiting for any resource, $E(WQ(\ell_a)) = E(WQ(\ell_b)) = E(WQ(\ell_c)) = \mathcal{R}^w_{2,1}$.

Before describing the techniques that govern when requests should be satisfied, we define relevant notation. We say that two resources ℓ_a and ℓ_b are *read shared*, denoted $\ell_a \sim \ell_b$,[1] if both ℓ_a and ℓ_b could be requested together as part of a single read request (*i.e.*, for some $\mathcal{R}^r_{i,k}$, $\{\ell_a, \ell_b\} \subseteq \mathcal{N}_{i,k}$). We call the set of all resources that are read shared with ℓ_a the *read set* of ℓ_a, denoted $S(\ell_a) = \{\ell_b | \ \ell_b \sim \ell_a\}$.

Example (cont'd) In Fig. 2, $\mathcal{R}^r_{5,1}, \mathcal{N}_{5,1} = \{\ell_a, \ell_b\}$. Thus, $\ell_a \sim \ell_b$ (and $\ell_b \sim \ell_a$). Since $\mathcal{R}^r_{5,1}$ is the only request for multiple resources, $S(\ell_a) = \{\ell_a, \ell_b\}$ and $S(\ell_c) = \{\ell_c\}$.

To avoid inconsistent phases, a write request may be forced to request additional resources besides those needed in its critical section. To reflect this, we let $\mathcal{D}_{i,k}$ denote the actual set of resources that $\mathcal{R}_{i,k}$ pertains to. For a read request $\mathcal{R}^r_{i,k}$, $\mathcal{D}_{i,k}$ is simply $\mathcal{N}_{i,k}$. However, for a write request $\mathcal{R}^w_{i,k}$, we define $\mathcal{D}_{i,k} = \bigcup_{\ell_a \in \mathcal{N}_{i,k}} S(\ell_a)$. While forcing write requests to acquire more resources than actually needed reduces concurrency, it does not affect asymptotic optimality. As we shall see, this expansion of write requests enables us to avoid inconsistent phases. Additionally, we

[1]Read sharing is reflexive and symmetric.



(a) Visual depiction of the schedule in the running example.

time	queue states			
	$RQ(\ell_a)$	$WQ(\ell_a)$	$RQ(\ell_b)$	$WQ(\ell_b)$
$[0, 2)$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$[2, 7)$	$\emptyset$	$\{\mathcal{R}^w_{2,1}\}$	$\emptyset$	$\{\mathcal{R}^w_{2,1}\}$
$[7, 8)$	$\emptyset$	$\{\mathcal{R}^w_{2,1}\}$	$\{\mathcal{R}^r_{5,1}\}$	$\{\mathcal{R}^w_{2,1}\}$
$[8, 10)$	$\emptyset$	$\emptyset$	$\{\mathcal{R}^r_{5,1}\}$	$\emptyset$
$[10, 12]$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

(b) Queue states over time corresponding to the schedule in (a).

Figure 2: Illustration of the running example.

shall show later that this expansion of write requests can be relaxed to enable additional concurrency on average.

Example (cont'd). Suppose $\mathcal{R}^w_{2,1}$ in Fig. 2 only needs $\mathcal{N}_{2,1} = \{\ell_a, \ell_c\}$ in its critical section. However, because $\ell_a \sim \ell_b$ and $\ell_a \in \mathcal{N}_{2,1}$, $\mathcal{R}^w_{2,1}$ requests $\mathcal{D}_{2,1} = \{\ell_a, \ell_b, \ell_c\}$.

General rules. The first few rules of the RSM are common to both readers and writers and describe the necessary actions that must be taken when a job either issues a request or completes a critical section.

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

G2 When $\mathcal{R}_{i,k}$ is satisfied, it is dequeued from either $RQ(\ell_a)$ (if it is a read request) or $WQ(\ell_a)$ (if it is a write request) for each $\ell_a \in \mathcal{D}_{i,k}$.

G3 When $\mathcal{R}_{i,k}$ completes, it unlocks all resources in $\mathcal{D}_{i,k}$.

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 (cont'd). At time $t = 8$, when $\mathcal{R}^r_{3,1}$ completes its critical section, $\mathcal{D}_{3,1} = \{\ell_c\}$ is unlocked. This allows $\mathcal{R}^w_{2,1}$ to be satisfied (as explained later), and therefore $\mathcal{R}^w_{2,1}$ is dequeued from $WQ(\ell_a)$, $WQ(\ell_b)$, and $WQ(\ell_c)$.

The remaining read- and write-specific rules rely on the concept of *entitlement*. Intuitively, a request becomes *entitled* once it is the next request to be satisfied (w.r.t. the resources for which it is waiting), and remains entitled until it is satisfied. While a request is entitled, it blocks all conflicting requests. Entitlement is differently defined for read and write requests, which is key to obtaining phase-fairness. We begin with read requests, which are entitled if they are blocked only by satisfied (and not entitled) write requests.

Def. 3. An unsatisfied read request $\mathcal{R}^r_{i,k}$ becomes *entitled* when there exists $\ell_a \in \mathcal{D}_{i,k}$ that is write locked, and for each resource $\ell_a \in \mathcal{D}_{i,k}$, $E(WQ(\ell_a))$ is not entitled (see Def. 4). (Note that $E(WQ(\ell_a)) = \emptyset$ could hold. In this case, we consider $E(WQ(\ell_a))$ to be a "null" request that is not entitled.) $\mathcal{R}^r_{i,k}$ remains entitled until it is satisfied.

Of course, if a newly issued read request does not conflict with satisfied or entitled incomplete requests, then it is satisfied immediately (see Rule R1 below) and Def. 3 does not apply (only unsatisfied requests can be entitled).

Example (cont'd). At time $t = 8$, $\mathcal{R}^r_{5,1}$ is blocked by $\mathcal{R}^w_{2,1}$, which holds ℓ_a, ℓ_b, and ℓ_c. By Def. 3, $\mathcal{R}^r_{5,1}$ becomes entitled at time $t = 8$ because ℓ_a is write locked and $E(WQ(\ell_a)) = E(WQ(\ell_b)) = \emptyset$.

Next we consider the writer case. Intuitively, an entitled write is the head of all relevant write queues and not blocked by any entitled (*i.e.*, only blocked by satisfied) reads.

Def. 4. An unsatisfied write request $\mathcal{R}^w_{i,k}$ becomes *entitled* when for each $\ell_a \in \mathcal{D}_{i,k}$, $\mathcal{R}^w_{i,k} = E(WQ(\ell_a))$, no read request in $RQ(\ell_a)$ is entitled (see Def. 3), and ℓ_a is not write locked. $\mathcal{R}^w_{i,k}$ remains entitled until it is satisfied.

Observe that an entitled write request $\mathcal{R}^w_{i,k}$ is only blocked by satisfied, but incomplete read requests since according to Def. 4 no resource in $\mathcal{D}_{i,k}$ is write locked.

Example (cont'd). At time $t = 5$, $\mathcal{R}^r_{3,1}$ holds ℓ_c, and blocks $\mathcal{R}^w_{2,1}$, which is waiting for ℓ_a, ℓ_b, and ℓ_c. Because $\mathcal{R}^w_{2,1}$ is the earliest timestamped writer waiting any of the resources, and none is write locked, $\mathcal{R}^w_{2,1}$ becomes entitled. Note that, although $\mathcal{R}^w_{2,1}$ is entitled, it is still blocked. Prior to $t = 5$, $\mathcal{R}^w_{3,1}$ was not be entitled because ℓ_a and ℓ_b were write locked by $\mathcal{R}^w_{1,1}$.

An entitled request (read or write) may be blocked by multiple requests, each holding different resources. We let $B(\mathcal{R}_{i,k}, t)$ be the set of satisfied requests that conflict with an entitled request $\mathcal{R}_{i,k}$ at time t (*i.e.*, the set of requests that block $\mathcal{R}_{i,k}$ at time t). Note that since read requests do not conflict with each other, $B(\mathcal{R}^r_{i,k}, t)$ only contains write requests. Analogously, as pointed out above, an entitled writer is only blocked by read requests, and thus $B(\mathcal{R}^w_{i,k}, t)$ only consists of read requests. This matches the intuition that reads concede to writes, and writes concede to reads.

Example (cont'd). At any time $t \in [6, 8)$, $\mathcal{R}^w_{2,1}$ is blocked by $\mathcal{R}^r_{3,1}$, thus $B(\mathcal{R}^w_{2,1}, t) = \{\mathcal{R}^r_{3,1}\}$. Earlier, at any time $t \in [5, 6)$, $\mathcal{R}^w_{2,1}$ is blocked by both $\mathcal{R}^r_{3,1}$ and $\mathcal{R}^r_{4,1}$, thus $B(\mathcal{R}^w_{2,1}, t) = \{\mathcal{R}^r_{3,1}, \mathcal{R}^r_{4,1}\}$.

Reader rules. We next define reader-specific rules, which utilize the previously given definition of entitlement. These rules define the behavior of the RSM, when a read request is issued and satisfied, respectively.

R1 When $\mathcal{R}^r_{i,k}$ is issued, for each $\ell_a \in \mathcal{D}_{i,k}$, $\mathcal{R}^r_{i,k}$ is enqueued in $RQ(\ell_a)$. If $\mathcal{R}^r_{i,k}$ does not conflict with any entitled or satisfied write requests, then it is satisfied immediately.

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

Example (cont'd). At time $t = 3$, $\mathcal{R}^r_{3,1}$ is issued and it is satisfied immediately by Rule R1. $\mathcal{R}^r_{3,1}$ is allowed to "cut ahead" of $\mathcal{R}^w_{2,1}$ in this case because $\mathcal{R}^w_{2,1}$ is not entitled, and ℓ_c is unlocked. Further, at time $t = 10$, $\mathcal{R}^r_{5,1}$ is satisfied by Rule R2. This is because $\mathcal{R}^r_{5,1}$ is entitled, and $\mathcal{R}^w_{2,1}$ completed it critical section, which unlocked ℓ_a and ℓ_b.

Writer rules. The writer rules parallel the reader rules.

W1 When $\mathcal{R}^w_{i,k}$ is issued, for each $\ell_a \in \mathcal{D}_{i,k}$, $\mathcal{R}^w_{i,k}$ is enqueued in timestamp order in the write queue $WQ(\ell_a)$. If $\mathcal{R}^w_{i,k}$ does not conflict with any entitled or satisfied requests (read or write), then it is satisfied immediately.

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

Full example. At time $t = 1$, a write request $\mathcal{R}^w_{1,1}$ is issued for ℓ_a and ℓ_b, which is immediately satisfied (by Rule W1). At time $t = 2$, another write request, $\mathcal{R}^w_{2,1}$ is issued for ℓ_a, ℓ_b, and ℓ_c, which is enqueued in $WQ(\ell_a)$, $WQ(\ell_b)$, and $WQ(\ell_c)$ (by Rule W1). $\mathcal{R}^r_{3,1}$ is issued and satisfied immediately at time $t = 3$ by Rule R1, as previously described. Similarly, at time $t = 4$, $\mathcal{R}^r_{4,1}$ is issued and satisfied immediately (by Rule R1). Note that at time $t = 4$, both $\mathcal{R}^r_{3,1}$ and $\mathcal{R}^r_{4,1}$ have read locked ℓ_b, demonstrating reader parallelism. Further, at time $t = 4$, ℓ_a and ℓ_b are write locked while ℓ_c is read locked, a level of concurrency only possible with fine-grained locking. When $\mathcal{R}^w_{1,1}$ completes at time $t = 5$, $\mathcal{R}^w_{2,1}$ becomes entitled. At time $t = 7$, $\mathcal{R}_{5,1}$ is issued for ℓ_b and ℓ_c, but it is not satisfied because $\mathcal{R}^w_{2,1}$ is entitled to both resources. After $\mathcal{R}^r_{3,1}$ completes at time $t = 8$, $\mathcal{R}^w_{2,1}$ is satisfied (by Rule W2). Finally, after $\mathcal{R}^w_{2,1}$ completes at time $t = 10$, $\mathcal{R}^r_{5,1}$ is satisfied (by Rule R2).

This concludes the definition and introduction of the R/W RNLP. To summarize, the R/W RNLP implements phase-fairness, where reads concede to writes and writes concede to reads. To avoid the problem of inconsistent phases (which could otherwise arise due to interleaved reads and writes), we have introduced the concept of entitled blocking. Intuitively, an entitled request is "next in line" with regard to the requested resources and only blocked by satisfied, but incomplete requests of the opposite kind.

3.3 Analysis

In this subsection, we present blocking analysis for the R/W RNLP. We begin by establishing several properties pertaining to entitlement and request satisfaction (Lemmas 2-4), which follow from the previous rules and definitions, and are needed to establish Cors. 1 and 2 These corollaries are central to our blocking analysis.

Let I be an invocation of the locking protocol (read or write issuance or read or write completion) at time t_I, and let $t_I^- = \lim_{\epsilon \to 0} t_I - \epsilon$ be the time instant immediately prior to that invocation. We say that I *entitles* (*satisfies*) a request

$\mathcal{R}_{i,k}$ if $\mathcal{R}_{i,k}$ becomes entitled (satisfied) as a result of I (*i.e.*, $\mathcal{R}_{i,k}$ is entitled (satisfied) after I but not before I).

Lemma 2. *The following properties of satisfaction and entitlement hold.*

- ***E1*** *If I satisfies $\mathcal{R}^r_{i,k}$, then I is either a read issuance or a write completion.*
- ***E2*** *If I satisfies $\mathcal{R}^w_{i,k}$, then I is either a write issuance, a read completion, or a write completion.*
- ***E3*** *If I satisfies $\mathcal{R}^r_{i,k}$ and I is the issuance of read request $\mathcal{R}^r_{x,y}$, then $\mathcal{R}^r_{i,k} = \mathcal{R}^r_{x,y}$.*
- ***E4*** *If I satisfies $\mathcal{R}^w_{i,k}$ and I is the issuance of write request $\mathcal{R}^w_{x,y}$, then $\mathcal{R}^w_{i,k} = \mathcal{R}^w_{x,y}$.*
- ***E5*** *If I satisfies $\mathcal{R}^w_{i,k}$ and I is the completion of a conflicting read request $\mathcal{R}^r_{x,y}$, then at time t_I^-, $\mathcal{R}^w_{i,k}$ is entitled, and $B(\mathcal{R}^w_{i,k}, t_I^-) = \{\mathcal{R}^r_{x,y}\}$.*
- ***E6*** *If I satisfies $\mathcal{R}^r_{i,k}$ and I is the completion of a conflicting write request $\mathcal{R}^w_{x,y}$, then at time t_I^-, $\mathcal{R}^r_{i,k}$ is entitled, and $B(\mathcal{R}^r_{i,k}, t_I^-) = \{\mathcal{R}^w_{x,y}\}$.*
- ***E7*** *If I satisfies $\mathcal{R}^w_{i,k}$ and I is the completion of a conflicting write request $\mathcal{R}^w_{x,y}$, then at time t_I^-, for each $\ell_a \in \mathcal{D}^w_{i,k}$, $\mathcal{R}^w_{i,k} = E(WQ(\ell_a))$ and no read request in $RQ(\ell_a)$ is entitled, and for each resource $\ell_a \in \mathcal{D}_{i,k}$, ℓ_a is either locked by $\mathcal{R}^w_{x,y}$, or unlocked.*
- ***E8*** *If I entitles $\mathcal{R}^r_{i,k}$, then I is a read issuance or a read completion.*
- ***E9*** *If I entitles $\mathcal{R}^w_{i,k}$, then I is a write issuance or a write completion.*
- ***E10*** *If $\mathcal{R}^w_{i,k}$ and $\mathcal{R}^r_{x,y}$ conflict, then they are not simultaneously entitled.*

Proof. We prove the stated properties in succession.

Prop. E1. If I is a write issuance, then it releases no resources for which $\mathcal{R}^r_{i,k}$ is waiting, and hence cannot cause $\mathcal{R}^r_{i,k}$ to become satisfied. On the other hand, if I is a read completion and $\mathcal{R}^r_{i,k}$ is not entitled prior to I, then by Rule R2, I cannot cause $\mathcal{R}^r_{i,k}$ to become satisfied. If I is a read completion and $\mathcal{R}^r_{i,k}$ is entitled (and hence blocked) prior to I, then $B(\mathcal{R}^r_{i,k}, t_I^-)$ contains at least one write request; I cannot cause this write request to complete, thus following I, $\mathcal{R}^r_{i,k}$ remains entitled (and hence blocked).

Prop. E2. Like the first case considered above, I cannot cause $\mathcal{R}^w_{i,k}$ to be become satisfied if it is a read issuance.

Prop. E3. If I is the issuance of read request $\mathcal{R}^r_{i,k}$, then it does not unlock any resources, and hence cannot cause any previously issued request to become satisfied. However, by Rule R1, I may cause $\mathcal{R}^r_{i,k}$ itself to become satisfied.

Prop. E4. Follows similarly to Prop. E3.

Prop. E5. By Rule W2, if I satisfies $\mathcal{R}^w_{i,k}$, then prior to I, $\mathcal{R}^w_{i,k}$ must have been entitled, and $\mathcal{R}^r_{x,y}$ must have been the only request that blocked $\mathcal{R}^w_{i,k}$.

Prop. E6. Follows similarly to Prop. E5 (using Rule R2).

Prop. E7. By Rule W2, if I satisfies $\mathcal{R}^w_{i,k}$, then it must be entitled. However, because $\mathcal{R}^w_{x,y}$ is satisfied at time t_I^- and conflicts with $\mathcal{R}^w_{i,k}$, then $\mathcal{R}^w_{i,k}$ is not entitled at time t_I^- by Def. 4. For $\mathcal{R}^w_{i,k}$ to be satisfied at time t_I, by Rule W2, it must become entitled at time t_I. By Def. 4 for $\mathcal{R}^w_{i,k}$ to be entitled at time t_I, after $\mathcal{R}^w_{x,y}$ unlocks all resources in $\mathcal{D}_{x,y}$, for each $\ell_a \in \mathcal{D}_{i,k}$, $\mathcal{R}^w_{i,k} = E(WQ(\ell_a))$, and no read request in $RQ(\ell_a)$ is entitled, and ℓ_a is not write locked. Furthermore, since $\mathcal{R}^w_{i,k}$ is satisfied at time t_I, then all resources in $\mathcal{D}_{i,k}$ are unlocked after $\mathcal{R}^w_{x,y}$ completes. The claim follows.

Prop. E8. By Def. 3, if $\mathcal{R}^r_{i,k}$ is unsatisfied and not entitled prior to I, *i.e.*, at time t_I^-, then it is blocked at t_I^- by an entitled write request, $\mathcal{R}^w_{x,y}$. Thus, by Def. 4, the following hold at time t_I^-: $\mathcal{R}^w_{x,y}$ is at the head of each write queue in which it is enqueued; no resource for which $\mathcal{R}^w_{x,y}$ is waiting is write locked; and $\mathcal{R}^w_{x,y}$ is not blocked by any entitled read request. Recall that entitled requests are, by definition, unsatisfied. Thus, $\mathcal{R}^w_{x,y}$ must be blocked by at least one *satisfied* read request at t_I^-. Now, if I is a write issuance, then $\mathcal{R}^w_{x,y}$ clearly remains entitled at t_I, and hence $\mathcal{R}^r_{i,k}$ is not entitled at t_I. On the other hand, if I is a write completion, then it may cause certain entitled reads to become satisfied; however, it will not cause the satisfied read that blocks $\mathcal{R}^w_{x,y}$ to complete. Thus, as before, $\mathcal{R}^w_{x,y}$ remains entitled at t_I, and hence $\mathcal{R}^r_{i,k}$ is not entitled at t_I.

Prop. E9. Follows similarly to Prop. E8.

Prop. E10. Defs. 3 and 4 preclude conflicting read and write requests from both becoming entitled due to *separate* invocations of the locking protocol. Props. E8 and E9 preclude such requests from both becoming entitled due to the *same* invocation of the locking protocol. □

Next we show that once a write request $\mathcal{R}^w_{i,k}$ is entitled, no conflicting request $\mathcal{R}_{x,y}$ can be satisfied before it, which implicitly bounds how long it remains entitled.

Lemma 3. *If a write request $\mathcal{R}^w_{i,k}$ is entitled before and after I and $\mathcal{R}_{x,y} \in B(\mathcal{R}^w_{i,k}, t_I)$, then $\mathcal{R}_{x,y} \in B(\mathcal{R}^w_{i,k}, t_I^-)$.*

Proof. Suppose not. Then the mentioned request $\mathcal{R}_{x,y}$ (read or write) is satisfied by I, and by the definition of $B(\mathcal{R}^w_{i,k}, t_I)$, $\mathcal{R}_{x,y}$ conflicts with $\mathcal{R}^w_{i,k}$.

Assume that $\mathcal{R}^r_{x,y}$ is a read request. Then, by Prop. E1, I is a read issuance or a write completion. If I is a read issuance, then by Prop. E3, $\mathcal{R}^r_{x,y}$ is issued at t_I; however, by Rule R1, I cannot then satisfy $\mathcal{R}^r_{x,y}$ because $\mathcal{R}^w_{i,k}$ is entitled. If I is a write completion, then by Prop. E6, $\mathcal{R}^r_{x,y}$ is entitled at t_I^-; however, by Prop. E10, this implies that $\mathcal{R}^w_{i,k}$ is not entitled at t_I^-, contradicting the lemma statement.

Now assume that $\mathcal{R}^w_{x,y}$ is a write request. Then, by Prop. E2, I is a write issuance, read completion, or write completion. If I is a write issuance or read completion, then we can derive a contradiction via reasoning similar to that above (but using Prop. E4, Rule W1, and Prop. E5 together with Prop. E10). So, suppose that I is a write completion. By the statement of the lemma, it follows that $\mathcal{R}^w_{i,k}$ and $\mathcal{R}^w_{i,k}$ conflict and share some resource ℓ_c. Moreover, by Prop. E7,

$\mathcal{R}^w_{x,y} = E(WQ(\ell_c))$ holds at t_I^-. However, by Def. 4, this contradicts the assumption that $\mathcal{R}^w_{i,k}$ is entitled at t_I^-. □

Corollary 1. Suppose that the request $\mathcal{R}^w_{i,k}$ becomes entitled at time t_e and satisfied at time t_s. Then, no new requests may be added to $B(\mathcal{R}^w_{i,k}, t)$ at any time time $t \in [t_e, t_s)$.

Similar to Lemma 3, we next show that once a read request $\mathcal{R}^r_{i,k}$ becomes entitled, no conflicting request can be satisfied before it.

Lemma 4. *If a read request $\mathcal{R}^r_{i,k}$ is entitled before and after I and $\mathcal{R}^w_{x,y} \in B(\mathcal{R}^w_{i,k}, t_I)$, then $\mathcal{R}^w_{x,y} \in B(\mathcal{R}^w_{i,k}, t_I^-)$.*

Proof. Suppose not. Then, the mentioned write request $\mathcal{R}^w_{x,y}$ is satisfied by I, and by the definition of $B(\mathcal{R}^w_{i,k}, t_I)$, $\mathcal{R}^w_{x,y}$ conflicts with $\mathcal{R}^w_{i,k}$. Thus, by Prop. E2, I is either a write issuance, read completion, or write completion. If I is a write issuance, then by Prop. E4, I is the issuance of $\mathcal{R}^w_{x,y}$ itself; however, by Rule W1, I cannot satisfy $\mathcal{R}^w_{x,y}$, because $\mathcal{R}^r_{i,k}$ is entitled prior to I. If I is a read (resp., write) completion, then by Prop. E5 (resp., Prop. E7), $\mathcal{R}^w_{x,y}$ is entitled at t_I^-; however, by Prop. E10, this contradicts the assumption that $\mathcal{R}^r_{i,k}$ is entitled at t_I^-. □

Corollary 2. Suppose that the request $\mathcal{R}^r_{i,k}$ becomes entitled at time t_e and satisfied at time t_s. Then, no new requests may be added to $B(\mathcal{R}^r_{i,k}, t)$ at any time $t \in [t_e, t_s)$.

Below, we show that worst-case acquisition delay is $O(1)$ for readers and $O(m)$ for writers. The following lemmas are used in establishing these results.

Lemma 5. *A write request $\mathcal{R}^w_{i,k}$ experiences acquisition delay of at most L^r_{max} time units after becoming entitled.*

Proof. Suppose that $\mathcal{R}^w_{i,k}$ becomes entitled at time t_e and satisfied at t_s. By Cor. 1, new requests are not added to $B(\mathcal{R}^w_{i,k}, t)$ at any $t \in [t_e, t_s)$. Moreover, by Def. 4, each request in $B(\mathcal{R}^w_{i,k}, t)$ is a read. By Prop. P1, every request in $B(\mathcal{R}^w_{i,k}, t_e)$ is scheduled, and therefore will complete in at most L^r_{max} time units. Thus, by time $t_e + L^r_{max}$, $\mathcal{R}^w_{i,k}$ will not be blocked, and by Rule W2, will be satisfied. □

Lemma 6. *If $\mathcal{R}^w_{i,k}$ is the earliest timestamped write request among all incomplete write requests, then $\mathcal{R}^w_{i,k}$ is either satisfied or entitled.*

Proof. Suppose not. Then, by Def. 4, either (i) for some resource $\ell_a \in \mathcal{D}_{i,k}$, $\mathcal{R}^w_{i,k} \neq E(WQ(\ell_a))$, (ii) some request $\mathcal{R}^r_{x,y} \in RQ(\ell_a)$ is entitled, or (iii) ℓ_a is write locked. By Rule W1, (i) and (iii) are not possible since the write queues are timestamp ordered, and $\mathcal{R}^w_{i,k}$ is the earliest timestamped incomplete write request.

For (ii), assume $\mathcal{R}^r_{x,y}$ is entitled and $\ell_a \in \mathcal{D}_{i,k} \cap \mathcal{D}_{x,y}$. Then, by Def. 3, $\mathcal{R}^r_{x,y}$ is blocked by a satisfied write request $\mathcal{R}^w_{h,l}$. Recall that $\mathcal{R}^w_{h,l}$ must request all resources in the read sets of all resources in $\mathcal{N}_{h,l}$. Further, ℓ_a must be in at least one of these read sets. Thus, $\ell_a \in \mathcal{D}_{h,l} \cap \mathcal{D}_{i,k}$, and $\mathcal{R}^w_{h,l}$ and $\mathcal{R}^w_{i,k}$ conflict. Therefore, since $ts(\mathcal{R}^w_{i,k}) < ts(\mathcal{R}^w_{h,l})$, $\mathcal{R}^w_{h,l}$ cannot be satisfied. □

Theorem 1. *The worst-case acquisition delay of a read request $\mathcal{R}^r_{i,k}$ is at most $L^w_{max} + L^r_{max}$ time units.*

Proof. We first show that if $\mathcal{R}^r_{i,k}$ is issued at time t_i, then it must become entitled or satisfied by time $t_i + L^r_{max}$. Suppose not. Then, throughout the interval $[t_i, t_i + L^r_{max})$, $\mathcal{R}^r_{i,k}$ is blocked by a non-empty set W of conflicting entitled write requests, for otherwise, $\mathcal{R}^r_{i,k}$ would become entitled (by Def. 3) or satisfied (by Rule R1). By Prop. P1 and Lemma 5, each write request $\mathcal{R}^w_{x,y} \in W$ will be satisfied by time $t_i + L^r_{max}$. Once all such write requests are satisfied, by Def. 3, $\mathcal{R}^r_{i,k}$ will become entitled or satisfied, a contradiction.

If $\mathcal{R}^r_{i,k}$ becomes satisfied by time $t_i + L^r_{max}$, then its acquisition delay is at most L^r_{max} time units. Consider now the other possibility, *i.e.*, that $\mathcal{R}^r_{i,k}$ becomes entitled by some time $t_e \leq t_i + L^r_{max}$. In this case, we show that $\mathcal{R}^r_{i,k}$ is satisfied by time $t_e + L^w_{max}$, from which an acquisition delay of at most $L^r_{max} + L^w_{max}$ time units follows. By Cor. 2, the number of resource-holding write requests blocking $\mathcal{R}^r_{i,k}$ monotonically decreases until $\mathcal{R}^r_{i,k}$ is satisfied. By Prop. P1, each such blocking request completes in at most L^w_{max} time units. Thus, $\mathcal{R}^r_{i,k}$ is satisfied by time $t_e + L^w_{max}$. □

Theorem 2. *The worst-case acquisition delay of a write request $\mathcal{R}^w_{i,k}$ is at most $(m-1)(L^r_{max} + L^w_{max})$ time units.*

Proof. Suppose that the write request $\mathcal{R}^w_{i,k}$ is issued at time t_i and not satisfied immediately. Let $\mathcal{R}^w_{x,y}$ be the incomplete write request with the earliest timestamp at t_i ($\mathcal{R}^w_{x,y}$ could be $\mathcal{R}^w_{i,k}$). By Lemma 6, $\mathcal{R}^w_{x,y}$ is either entitled or satisfied at t_i. Suppose the latter is true, *i.e.*, $\mathcal{R}^w_{x,y}$ is satisfied at t_i. Then, by Prop. P1, $\mathcal{R}^w_{x,y}$ completes its critical section by time $t_i + L^w_{max}$. By Prop. P2, there are at most $m-1$ incomplete write requests with timestamps earlier than that of $\mathcal{R}^w_{i,k}$ at t_i. Thus, by time $t_i + L^w_{max}$, there are at most $m-2$ such requests. By Lemmas 5 and 6, the one with the earliest timestamp is satisfied by time $t_i + L^w_{max} + L^r_{max}$, and thus, by Prop. P1, completes its critical section by time $t_i + L^w_{max} + L^r_{max} + L^w_{max}$. Continuing inductively, all earlier-timestamped write requests complete their critical sections by time $t_i + L^w_{max} + (m-2)(L^r_{max} + L^w_{max})$. At that time, $\mathcal{R}^w_{i,k}$ has the earliest timestamp. Hence, by Lemma 5, it is satisfied by time $t_i + L^w_{max} + (m-2)(L^w_{max} + L^r_{max}) + L^r_{max}$, *i.e.*, $\mathcal{R}^w_{i,k}$'s acquisition delay is at most $(m-1)(L^r_{max} + L^w_{max})$ time units.

The remaining possibility to consider is that $\mathcal{R}^w_{x,y}$ is entitled at t_i. In this case, by Def. 4, $\mathcal{R}^w_{x,y}$ is blocked by some read request $\mathcal{R}^r_{h,l}$. Thus, by Prop. P2, there are at most $m-2$ incomplete write requests with timestamps earlier than that of $\mathcal{R}^w_{i,k}$ at t_i. Reasoning as above, it follows that $\mathcal{R}^w_{i,k}$'s acquisition delay is at most $(m-2)(L^r_{max} + L^w_{max}) + L^r_{max}$ time units. (Note that the blocking of $\mathcal{R}^w_{x,y}$ due to $\mathcal{R}^r_{h,l}$ is accounted for in this reasoning by Lemma 5.) □

For the case when waiting is realized by spinning (Rule S1), the worst-case acquisition delay for either reads or writes is the worst-case s-blocking (recall Def. 2). However, the non-preemptive spinning can cause other jobs, even non-resource-using jobs, to be pi-blocked (recall Def. 1) upon release. For example, if a high-priority job J_h is released that has sufficient priority to be scheduled, but a low-priority job J_l is spinning non-preemptively, then J_h is pi-blocked. The worst-case pi-blocking can easily be shown to be $m \cdot \max(L^w_{max}, L^r_{max})$ through analysis similar to single-resource spin-based mutex or reader-writer locks [4, 7]. In Sec. 4, we briefly describe techniques to reduce this $O(m)$ pi-blocking to $O(1)$.

In the remainder of this section, we describe additional features that can be incorporated into the R/W RNLP to improve performance in many cases. We present these features independently for ease of exposition, but note that they can be combined in a real implementation.

3.4 Requesting Fewer Resources

To solve the problem of inconsistent phases, we previously required writes to request a potentially expended set of resources. This property was necessary to ensure that the earliest timestamped write request was not blocked by a write request with a later timestamp (seen in Lemma 6). However, the additional resources requested do not need to be actually locked to ensure the proper satisfaction ordering.

Instead of forcing a request $\mathcal{R}^w_{i,k}$ to request additional non-needed resources, we propose to enqueue a *placeholder request* $\mathcal{R}^p_{i,k}$ in the queues of all non-needed resources that would have previously been requested, $\mathcal{M}_{i,k} = \bigcup_{\ell_a \in \mathcal{N}_{i,k}} S(\ell_a) \setminus \mathcal{N}_{i,k}$. Thus, $\mathcal{D}_{i,k} = \mathcal{N}_{i,k}$. In this case, the RSM functions as previously described with the following exceptions. A placeholder request is never entitled or satisfied. Instead, each placeholder request $\mathcal{R}^p_{i,k}$ is removed from the write queue in which it is enqueued when $\mathcal{R}^w_{i,k}$ becomes entitled or satisfied. Therefore, until $\mathcal{R}^w_{i,k}$ becomes entitled, its associated placeholder requests prevent later-issued write requests from becoming entitled or satisfied, thereby ensuring that Lemma 6 is not violated.

Example (cont'd). Continuing from the example in Fig. 2, consider that $\mathcal{R}^w_{1,1}$ only needed $\mathcal{N}_{1,1} = \{\ell_b\}$ and $\mathcal{R}^w_{2,1}$ only needed $\mathcal{N}_{2,1} = \{\ell_a, \ell_c\}$. When $\mathcal{R}^w_{1,1}$ is issued, it would enqueue a placeholder in $WQ(\ell_a)$, but since it is satisfied immediately, the placeholder is removed. When $\mathcal{R}^w_{2,1}$ is issued, it enqueues in $WQ(\ell_a)$ and $WQ(\ell_c)$, and enqueues a placeholder in $WQ(\ell_b)$. However, since $\mathcal{R}^w_{2,1}$ is not blocked by any conflicting requests, since $\mathcal{R}^w_{1,1}$ only holds the lock on ℓ_b, then $\mathcal{R}^w_{2,1}$ can be satisfied immediately at time $t = 2$, thereby improving concurrency.

Note that using placeholder requests, as compared to requesting non-needed resources, allows for additional concurrency. However, this parallelism is not reflected in the worst-case blocking bounds under our analysis assumptions. In future work, it may be possible to reflect the improved concurrency via more fine-grained blocking analysis, similar to that presented in [4, Chaps. 5-6].

3.5 R/W Mixing

In this subsection, we show that we can also improve concurrency by relaxing Assumption 1 to allow jobs to request read access to some resources and write access to others.

First, we extend our notation. We denote the set of resources that $\mathcal{R}_{i,k}$ needs read (write) access to as $\mathcal{N}^r_{i,k}$ ($\mathcal{N}^w_{i,k}$) and we let $\mathcal{N}_{i,k} = \mathcal{N}^r_{i,k} \cup \mathcal{N}^w_{i,k}$. If $\mathcal{N}^w_{i,k} = \emptyset$, then we say $\mathcal{R}^r_{i,k}$ is a read request, otherwise we say that $\mathcal{R}^w_{i,k}$ is a write request. With this notation, a mixed request is a write request $\mathcal{R}^w_{i,k}$ with $\mathcal{N}^r_{i,k} \neq \emptyset$ and $\mathcal{N}^w_{i,k} \neq \emptyset$. We also adapt our definition of the read shared relation, $\sim$. Given two resources ℓ_a and ℓ_b, we say that ℓ_b is read shared with ℓ_a, if for some potential request $\mathcal{R}_{i,k}$, $\ell_a \in \mathcal{N}_{i,k}$, and $\ell_b \in \mathcal{N}^r_{i,k}$.[2]

Next we describe how the rules of the RSM support such behavior with only a minor modification. Intuitively, a mixed request is treated almost exactly like an exclusively-write request, though there are three key differences. First, a mixed request can be satisfied if it is waiting for read access to a resource that is read locked. Second, when a mixed request is satisfied, resources for which read-only access is needed are read locked, not write locked, which allows read requests to be satisfied concurrently. Third, with respect to writer entitlement (Def. 4), blocked write requests treat a resource that is read-locked by a mixed request as if it were write locked.

To this end, when $\mathcal{R}^w_{i,k}$ is issued, it is enqueued in the write queue for each resource $\ell_a \in \mathcal{N}_{i,k}$. As described in the previous subsection, a placeholder request $\mathcal{R}^p_{i,k}$ is enqueued in each $WQ(\ell_a)$ for $\ell_a \in \mathcal{M}_{i,k}$. The definition of write entitlement (Def. 4) remains largely unchanged, with the minor exception that a write request does not become entitled if a resource that it requires is read-locked by a mixed request. We note, however, that for $\mathcal{R}^w_{i,k}$ to be entitled, it must be the head of each write queue in which it is enqueued, even for resources for which it does not require write access. The rest of the RSM remains unchanged, as the rules for satisfaction (Rules R1, R2, W1, W2) were originally specified in terms of conflicting requests. We next demonstrate such functionality via an example.

Example (cont'd). Consider the schedule depicted in Fig. 2, without the placeholder optimization described previously. Assume that $\mathcal{R}^w_{2,1}$ was actually a mixed request $\mathcal{R}^w_{2,1}$ that required read access to $\mathcal{D}^r_{2,1} = \{\ell_a, \ell_b\}$ and write access to $\mathcal{D}^w_{2,1} = \{\ell_c\}$. Then when $\mathcal{R}^r_{5,1}$ is issued at time $t = 7$, because it does not conflict with $\mathcal{R}^w_{2,1}$ (both requests only require read access to ℓ_a and ℓ_b), then $\mathcal{R}^r_{5,1}$ can be satisfied immediately by Rule R1.

3.6 R-to-W Upgrading

As mentioned previously, another desirable feature for R/W locking protocols and STM is the ability to upgrade a read request to a write request. In this subsection, we show how to support such functionality in the R/W RNLP.

We call a read request that can be upgraded to a write request an *upgradeable* request, and we denote such a request as $\mathcal{R}^u_{i,k}$. Intuitively, we treat an upgradeable request as a write request that can optimistically execute read-only code while the needed resources are read-locked to determine if write access is necessary. Since the blocking bounds of a write request assume that it will be blocked by other read requests, the optimistic execution of the read-only section

[2]The read sharing relation may not be symmetric with mixed requests.

essentially executes for free. Thus, an upgradeable request has the same worst-case blocking bounds as a write request, but may offer additional concurrency if the write segment of the critical section is not required.

To support this behavior in the RSM, we treat $\mathcal{R}^{u}_{i,k}$ as two separate requests, a read request,[3] $\mathcal{R}^{u_r}_{i,k}$ and a write request $\mathcal{R}^{u_w}_{i,k}$, which can cancel each other if necessary.[4] When $\mathcal{R}^{u}_{i,k}$ is issued, $\mathcal{R}^{u_r}_{i,k}$ is enqueued as a read request and $\mathcal{R}^{u_w}_{i,k}$ is enqueued as a write request. If $\mathcal{R}^{u_w}_{i,k}$ is satisfied before $\mathcal{R}^{u_r}_{i,k}$, then $\mathcal{R}^{u_r}_{i,k}$ is canceled and removed from all read queues. If $\mathcal{R}^{u_r}_{i,k}$ is satisfied first, it executes its critical section, and upon completion or realization that upgrading is not necessary, $\mathcal{R}^{u_w}_{i,k}$ is canceled and removed from all write queues in which it is enqueued. If $\mathcal{R}^{u}_{i,k}$ must be upgraded, then when the read-only segment of its critical section completes, all resources are unlocked. Later, when $\mathcal{R}^{u_w}_{i,k}$ is satisfied, the job can execute the write segment of its critical section. Note that the state of any read objects may change between $\mathcal{R}^{u_r}_{i,k}$ completing, and $\mathcal{R}^{u_w}_{i,k}$ being satisfied. Thus, $\mathcal{R}^{u_w}_{i,k}$ may need to re-read some data. If this behavior is unacceptable for a given application, it should instead issue a write request for all resources that could potentially be written.

3.7 Incremental locking

In this subsection, we show how the RSM can be adapted to allow jobs to incrementally request resources they use within their critical section, as described earlier. We assume that it is known a priori the set of all all resources that could possibly be requested in this incremental fashion. While this assumption may seem limiting, such information is necessary for many real-time locking protocols, such as the well-known priority ceiling protocol (PCP) [17].

To support this functionality, we initially treat $\mathcal{R}_{i,k}$ as if it were a request for all of the resources for which it could potentially lock incrementally. From Cors. 1 and 2, after $\mathcal{R}_{i,k}$ becomes entitled, no conflicting request can be satisfied before $\mathcal{R}_{i,k}$. Thus, if $\mathcal{R}_{i,k}$ only initially requires access to some subset $s \subseteq \mathcal{D}_{i,k}$, it can be granted access as soon as it is entitled and each resource $\ell_a \in s$ is not locked by a conflicting request. If $\mathcal{R}_{i,k}$ later needs some additional resource(s) $s' \subseteq \mathcal{D}_{i,k} \setminus s$, then it waits until each $\ell_a \in s'$ is not locked by a conflicting request. However, because $\mathcal{R}_{i,k}$ is entitled to all resources in $\mathcal{D}_{i,k}$, the total duration of acquisition delay across all incremental requests is at most the worst-case acquisition delay previously proven.

We note that entitlement serves a similar purpose as the priority ceiling in the PCP [17], in that it prevents later-issued requests from acquiring resources that may be incrementally requested.

3.8 Suspension-based R/W RNLP

Before presenting the suspension-based variant of the R/W RNLP, we begin by explaining how we analyze the pi-blocking caused by suspension-based locking protocols. In recent work, Brandenburg and Anderson [5] gave two alternate definitions of pi-blocking for suspension-based systems, *suspension aware* (*s-aware*) and *suspension oblivious* (*s-oblivious*).

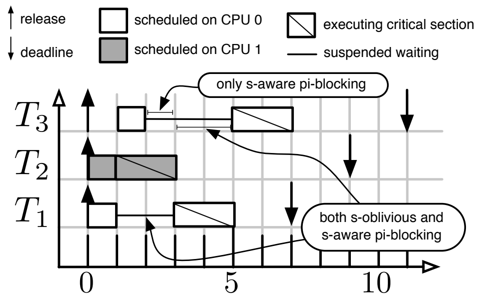


Figure 3: Illustration adapted from [5] of the difference between s-oblivious and s-aware analysis. In this example, three EDF-scheduled jobs share a single resource ℓ_a on two processors. During $[2, 4)$, J_3 is blocked, but there are m jobs with higher priority, thus J_3 is not s-oblivious pi-blocked. However, because J_1 is also suspended, J_3 is s-aware pi-blocked.

Def. 5. Under **s-aware** (**s-oblivious**) schedulability analysis, a job J_i incurs *s-aware* (*s-oblivious*) *pi-blocking* at time t if J_i is pending but not scheduled and fewer than c higher-priority jobs are **ready** (**pending**) in T_i's cluster.

The difference between s-oblivious and s-aware pi-blocking is demonstrated in Fig. 3. S-oblivious pi-blocking analysis is motivated by the fact that most multiprocessor schedulability tests do explicitly account for task suspensions. Instead, the suspensions of the highest priority jobs are analytically considered computation. This is modeled by adding the worst-case blocking term b_i to e_i. While this assumption is safe (it will not cause the task system to be incorrectly deemed schedulable), it can be pessimistic.

For systems for which there exist schedulability tests that explicitly incorporate suspension, (*i.e.*, s-aware schedulability tests), locking protocols can be analyzed under s-aware pi-blocking analysis. However, the vast majority of existing schedulability tests are not s-aware, as suspensions are notoriously difficult to analyze. Furthermore, there are many open problems concerning locking protocols that are optimal under s-aware analysis. Most relevant to this work is that no known R/W locking protocol is optimal under s-aware analysis, even for single-resource requests. However, recent experimental work has shown optimal s-oblivious locking protocols to be competitive with s-aware ones, even on systems for which there exists s-aware schedulability analysis. For these reasons, we assume s-oblivious analysis for all of our suspension-based results.

Progress mechanism for the suspension-based RSM. For suspension-based locks, we use *priority donation* [6], as the progress mechanism, instead of Rule S1. Below, we show that priority donation implies Properties P1 and P2, and can therefore be used with the RSM.

Intuitively, priority donation works by forcing high-

[3]We assume the worst-case execution time of the read-only segment of the upgradeable request finishes in L^{r}_{max} time

[4]With respect to Prop. P2, an upgradeable request is only one request.

priority jobs, upon release, to *donate* their priority to low-priority jobs with incomplete resource requests. Unlike priority inheritance, donation forms a static donation relationship that persists until the donee completes its critical section, or until an even higher priority job donates its priority to the donee instead. Priority donation therefore ensures that all resource-holding jobs are scheduled, and that the acquisition delay is bounded. With this understanding of donation, we prove that donation satisfies Prop. P1 and P2 via reference to the formal definition of donation [6].

Lemma 7. *Priority donation implies Properties P1 and P2.*

Proof. Prop. P1 exactly matches Prop. P1 of [6]. Prop. P2 follows directly from Lemma 3 in [6]. □

From Prop. P2 of [6], the worst-case duration of s-oblivious pi-blocking caused by priority donation, which effects all tasks in the system, is the worst-case acquisition delay plus the maximum critical section length. Therefore, from Theorems 1 and 2, the worst-case duration of priority donation is $L^w_{max} + (m-1)(L^r_{max} + L^w_{max}) = O(m)$. In Sec. 4, we briefly describe how this can be reduced to $O(1)$.

4 Conclusions

We have presented the R/W RNLP, which is the first fine-grained locking protocol for real-time multiprocessor systems that supports reader/writer sharing. The need to support two different kinds of operations on resources—reads and writes—introduces considerable difficulty in designing an asymptotically optimal locking protocol. As our design of the R/W RNLP evolved, we devised numerous mechanisms that at first seemed promising for enabling efficient reading (writing) only to later discover that optimality for writing (reading) had been compromised. Most of these mechanisms proved problematic because they did not adequately resolve the problem of inconsistent phases (noted earlier) in all cases.

One unfortunate side effect of the progress mechanisms considered in this paper is that they induce $O(m)$ per-job pi-blocking, even on jobs that do not share resources (though the same criticism applies to all known optimal R/W locks). However, in recent work, Brandenburg developed a new progress mechanism called *migratory priority inheritance* (*MPI*), which can be combined with priority donation to reduce per-job pi-blocking to $O(1)$ [8]. The main idea is to use priority donation for read requests and MPI for write requests. Due to space constraints, we were unable to incorporate this idea into the version of the R/W RNLP presented in this paper.

We also did not have sufficient space to present experimental results. In future work, we intend to implement the R/W RNLP and compare it to other sharing alternatives on the basis of real-time schedulability with measured overheads considered. Among such alternatives, we are especially interested in comparing against non-blocking techniques as used in non-blocking STM. As alluded to in the introduction, we intend to evolve the design of the R/W RNLP and use it as the basis for building a lock-based STM framework. From an analysis point of view, our main focus in this paper has been worst-case pi-blocking. In worst-case sharing scenarios, the only potential parallelism is among readers, and this is reflected in our blocking bounds. More information about sharing patterns is required to derive bounds that reflect parallelism among writers. We leave the derivation of such bounds as future work as well.

Acknowledgment: We thank Björn Brandenburg for insightful discussions about phase-fair locks and MPI, which helped in the design of the R/W RNLP.

References

[1] J. Anderson, R. Jain, and S. Ramamurthy. Implementing hard real-time transactions on multiprocessors. In *RTDB '97*.

[2] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. In *RTSS '95*.

[3] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *RTCSA '07*, 2007.

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

[5] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *RTSS '10*.

[6] B. Brandenburg and J. Anderson. Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In *EMSOFT '11*.

[7] B. Brandenburg and J. Anderson. Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-Time Systems Journal*, 46:25–87, Sept. 2010.

[8] B. Brandenburg and A. Bastoni. The case for migratory priority inheritance in linux: Bounded priority inversions on multiprocessors. In *RTLWS '12*.

[9] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *RTAS '08*.

[10] A. Dragojević and T. Harris. STM in the small: Trading generality for performance in software transactional memory. In *EuroSys '12*.

[11] M. El-Shambakey and Binoy Ravindran. STM concurrency control for embedded real-time software with tighter time bounds. In *DAC '12*.

[12] S. Fahmy, B. Ravindran, and E. Jensen. Response time analysis of software transactional memory-based distributed real-time systems. In *SAC '09*.

[13] M. Herlihy and J. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93*.

[14] F. Meawad, K Iyer, M. Schoeberl, and J. Vitek. Real-time wait-free queues using micro-transactions. In *JTRES '12*.

[15] T. Sarni, A. Queudet, and P. Valduriez. Real-time support for software transactional memory. In *RTCSA '09*.

[16] M. Schoeberl, F. Brandner, and Jan Vitek. RTTM: Real-time transactional memory. In *SAC '10*.

[17] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Trans. on Computers*, 39(9):1175–1185, Sep. 1990.

[18] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95*.

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