Multi-Resource Real-Time Reader/Writer Locks for Multiprocessors

Bryan C. Ward and James H. Anderson
Dept. of Computer Science, The University of North Carolina at Chapel Hill

***Abstract*—A *fine-grained* locking protocol permits multiple locks to be held simultaneously by the same task. 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. This variant is shown to have worst-case blocking no worse (and often better) than existing coarse-grained real-time reader/writer locking protocols, while allowing for additional parallelism. Experimental evaluations of the proposed protocol are presented that consider both schedulability (*i.e.*, the ability to validate timing constraints) and implementation-related overheads. These evaluations demonstrate that the RNLP (both the mutex and the proposed reader/writer variant) provides improved schedulability over existing coarse-grained locking protocols, and is practically implementable.**

I. Introduction

Real-time and safety-critical systems are becoming increasingly complex, demanding the use of multicore platforms to improve performance and decrease size, weight, and power (SWaP) consumption. Examples of such systems include next-generation unmanned aerial vehicles (UAVs), and autonomous vehicles. To leverage multicore platforms in such systems, multiprocessor scheduling and synchronization algorithms are required that enable real-time timing constraints to be provably satisfied. A large body of prior work has produced several viable scheduling algorithms. However, to fully harness the computational power of multicore platforms, new multiprocessor real-time synchronizations techniques are necessary to exploit the parallelism afforded by multicore chips.

While numerous synchronization algorithms have been proposed and developed for general-purpose computing environments, such algorithms are often not suitable for real-time systems. This is because in a real-time system the *worst-case* timing behavior of a synchronization algorithm must be analyzable to ensure that no deadlines will be missed on account of synchronization. Therefore, real-time synchronization algorithms are often designed to reduce the worst case, instead of improving the average case. Lock-based synchronization algorithms are particularly well suited to multiprocessor real-time systems, as their worst-case performance is often better and easier to analyze without excessive pessimism than non-blocking counterparts.

When locks are used to support resource sharing, two principal approaches exist: coarse-grained and fine-grained locking. Under *coarse-grained locking*, 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 called *group locking* [1], clearly limits 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.

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 [10]. The RNLP is actually a "pluggable" protocol that has different variants for different real-time schedulers and analysis assumptions. Most of these variants are asymptotically optimal with respect to worst-case priority-inversion blocking, or *pi-blocking* (see Sec. II). 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 previous work [3, 4, 5, 10], we judge the efficacy of a real-time locking protocol in terms of worst-case pi-blocking. We show that the R/W RNLP has worst-case pi-blocking no worse than previous coarse-grained reader/writer locking protocols [5]. To make the R/W RNLP easier to understand, we focus on a variant in which tasks block by *spinning* (busy waiting), though we note similar ideas can be used to construct a suspension-based lock.

Real-time vs. general-purpose performance evaluation. Unlike locks in general-purpose computing environments, where locks are evaluated based on metrics such as throughput or operations per unit time, real-time locks are evaluated on the basis of real-time *schedulability*, *i.e.*, the ability to validate timing correctness. Therefore, to demonstrate the performance gains provided by the R/W RNLP, we evaluated the schedulability of tens of thousands of randomly generated task systems using either the R/W RNLP or other alternatives. This study suggests that the improved parallelism afforded by fine-grained locking via the RNLP or R/W RNLP can be reflected in analysis to improve

Work supported by NSF grants CNS 1016954, CNS 1115284, CNS 1218693, and CNS 1239135; and ARO grant W911NF-09-1-0535.

schedulability. We also implemented a spin-based variant of the R/W RNLP and measured its lock and unlock overheads, which we found to be quite small. These results suggest that the R/W RNLP has practical merit.

Algorithmic and analytical challenges. The R/W RNLP was obtained by employing the concept of reader and writer "phases," as used in *phase-fair reader/writer* (*R/W*) *locks* [3, 5, 6], within the context of the RNLP [10], which provides only mutex sharing. The RNLP orders conflicting resource requests on a FIFO basis, *i.e.*, earlier requests are satisfied first. Thus, when abstractly considering behavior under the RNLP as a dynamically changing wait-for graph, an important *stability* property emerges: once a resource request is issued, its outgoing edge set, *i.e.*, the set of requests upon which it is waiting, does not change.

Phase-fair locks expressly violate this stability property. In order to enable $O(1)$ worst-case pi-blocking for read requests, phase-fair locks allow later-requested reads to "cut ahead" of earlier-requested writes. This is accomplished by alternating read and write phases; in a read (write) phase, the managed resource is accessed by *all* (*one*) issued read requests (write request). (Note that this is with respect to a *single* resource: prior work on phase-fair locks has not addressed the fine-grained sharing of multiple resources.) Because reads can "cut ahead" of writes, the outgoing edge set of write requests in the wait-for graph is not stable—in fact, it is not stable for any R/W locking protocol with $O(1)$ worst-case reader pi-blocking.

Dealing with this lack of stability was one of the main challenges we faced in designing the R/W RNLP since we desired $O(1)$ reader pi-blocking. One issue that arises on account of instability is what we call the *R/W ordering dilemma*. Consider a read request $\mathcal{R}_1^r$ that is waiting to access two resources, ℓ_a, which is read locked by request $\mathcal{R}_2^r$, and ℓ_b, which is write locked by request $\mathcal{R}_3^w$, as in Fig. 1 (*n.b.*, notation will be defined more rigorously in Sec. II). Subsequently, a write request $\mathcal{R}_4^w$ is issued for the read-locked resource ℓ_a. Which request should be satisfied first, the waiting read $\mathcal{R}_1^r$ or the waiting write $\mathcal{R}_4^w$ (*i.e.*, where should $\mathcal{R}_4^w$ be inserted into the wait-for graph)? Phase-fair logic suggests that $\mathcal{R}_4^w$ be satisfied first (left side of Fig. 1), as the resource for which it is currently waiting is read locked (*i.e.*, in a read phase, so a write phase should be next). However, this is problematic because it increases the blocking bound of the read request $\mathcal{R}_1^r$, which is already blocked by another writer ($\mathcal{R}_3^w$). Alternatively, if the read $\mathcal{R}_1^r$ is satisfied next (right side of Fig. 1), then the write request $\mathcal{R}_4^w$ may be blocked by two read requests, leading to longer pi-blocking bounds than under a phase-fair lock.

Because we desire $O(1)$ pi-blocking for read requests, we have no choice but to sometimes let read requests "cut ahead" of write requests when resolving the R/W ordering dilemma, as in phase-fair locks. As noted above, this "cutting ahead" inserts edges into the wait-for graph that are not in accordance with FIFO request ordering. This has an effect that is not just localized but system-wide: in the wait-for graph, entire *paths*, representing *transitive* blocking relationships, may be inconsistent with FIFO ordering. The resulting ***transitive*** *early-on-late pi-blocking* can be difficult to properly handle and analyze.

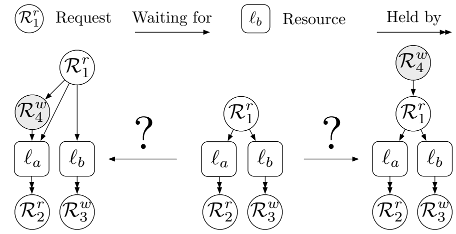


Figure 1: Illustration of the R/W ordering dilemma. The left and right side of the figure depict alternative places to insert a new write request $\mathcal{R}_4^w$ into the wait-for graph depicted in the middle.

Organization. After some preliminaries (Sec. II), we show how the above challenges can be addressed by presenting and analyzing the R/W RNLP (Sec. III). We then present our experimental evaluation and conclude (Secs. IV-V).

II. Background

We consider the commonly studied sporadic real-time task model [8], in which the system is composed of 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.) For notational simplicity, we assume that J_i issues at most one request, which we denote $\mathcal{R}_i$. 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*.

Real-time locking protocols must be coupled with a *progress mechanism* that ensures that lock-holding jobs "make progress." We require that such a mechanism satisfy the following two 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.

Progress mechanisms for suspension-based locks (*e.g.*, priority donation [4]) can be applied in the R/W RNLP, but such mechanisms are too complicated to describe in the space allowed. Therefore, for simplicity, we assume that jobs with incomplete resource requests execute non-preemptively (both while waiting via spinning and within their critical sections), which trivially satisfies these two properties.

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. 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. A task can migrate among the processors within its cluster. *Partitioned* and *global* scheduling are special cases of clustered scheduling, in which $c = 1$ and $c = m$, respectively. Additionally, we assume a job-level fixed priority scheduling algorithm, in which each job has a constant priority, but different jobs of the same task may have differing priorities.

Blocking. We evaluate the blocking of the presented locking protocols on the basis of their worst-case *priority-inversion blocking* (*pi-blocking*) [6].

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. Similarly to [3, 6, 10], for asymptotic analysis, we assume that m and n (the number of processors and tasks, respectively) are variables, and all other parameters are constants. Examples of such constants include critical section lengths. Additionally, we assume that locking-protocol invocations take zero time and all other overheads are negligible (such overheads can be easily factored into the final analysis [3, Chaps. 3,7]).

III. R/W RNLP

The goal of this paper is to extend the original mutex RNLP [10] to enable fine-grained reader/writer (R/W) sharing and, to the extent possible, enable non-conflicting requests to be satisfied concurrently. We also desire the following additional properties.

- **R/W mixing**. Some resources may be read and others written in one critical section. Such critical sections can be satisfied concurrently if they do not conflict.

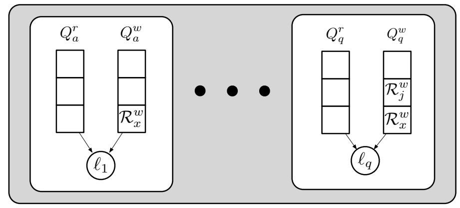


Figure 2: Queue structure in the R/W RNLP. For each resource $\ell_a \in \{\ell_1, \ldots, \ell_q\}$, there is a read queue Q_a^r and a write queue Q_a^w.

- **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 make the following simplifying assumption.

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 relax Assumption 1 later in this section and support the aforementioned three features. Until then (*i.e.*, while Assumption 1 is still in place), we use the following notation. We denote the set of resources that are needed in $\mathcal{R}_i$'s critical section as $\mathcal{N}_i$. 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 $\mathcal{R}_i^r$ ($\mathcal{R}_i^w$). We denote the longest read (write) critical section length as L_{max}^r (L_{max}^w), and we let $L_{max} = \max(L_{max}^r, L_{max}^w)$.

A. R/W RNLP

In the R/W RNLP, two queues are used per resource ℓ_a, a queue for readers, Q_a^r, and a queue for writers, Q_a^w, as depicted in Fig. 2. 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 $\mathcal{R}_i$ is recorded and denoted $ts(\mathcal{R}_i)$. 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 Q_a^w) as $E(Q_a^w)$. Similar to phase-fair locks [5], the queue from which requests are satisfied (Q_a^r or Q_a^w) alternates. The techniques that govern such alternation, however, are quite different from traditional phase-fair locks due to the R/W ordering dilemma.

Example. As we explain the rules of the R/W RNLP, we will reference relevant parts of the example schedule in Fig. 3, 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_c. At time $t = 2$, when $\mathcal{R}_2^w$ is issued, $ts(\mathcal{R}_2^w) = 2$ is established. Also, since $\mathcal{R}_2^w$ requires all three resources and since it is the only write request waiting for any resource, $E(Q_a^w) = E(Q_b^w) = E(Q_c^w) = \mathcal{R}_2^w$.

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 $\mathcal{R}_i^r$, $\{\ell_a, \ell_b\} \subseteq \mathcal{N}_i$). 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. 3, for $\mathcal{R}_5^r$, $\mathcal{N}_5 = \{\ell_a, \ell_b\}$. Thus, $\ell_a \sim \ell_b$ (and $\ell_b \sim \ell_a$). Since $\mathcal{R}_5^r$ is the only request for multiple resources, $S(\ell_a) = \{\ell_a, \ell_b\}$ and $S(\ell_c) = \{\ell_c\}$.

To avoid transitive early-on-late blocking, a write request may be forced to request additional resources besides those needed in its critical section. To reflect this, we let $\mathcal{D}_i$ denote the set of resources that $\mathcal{R}_i$ must actually request. For a read request $\mathcal{R}_i^r$, $\mathcal{D}_i$ is simply $\mathcal{N}_i$. However, for a write request $\mathcal{R}_i^w$, $\mathcal{D}_i = \bigcup_{\ell_a \in \mathcal{N}_i} S(\ell_a)$. While forcing write requests to acquire more resources than actually needed reduces runtime concurrency, it does not negatively affect worst-case pi-blocking. As we shall see, this expansion rule enables us to avoid transitive early-on-late blocking. Additionally, we shall show later that this expansion of write requests can be relaxed to enable additional concurrency on average.

Example (cont'd). Suppose $\mathcal{R}_2^w$ in Fig. 3 only needs $\mathcal{N}_2 = \{\ell_a, \ell_c\}$ in its critical section. Because $\ell_a \sim \ell_b$ and $\ell_a \in \mathcal{N}_2$, $\mathcal{R}_2^w$ actually requests $\mathcal{D}_2 = \{\ell_a, \ell_b, \ell_c\}$.

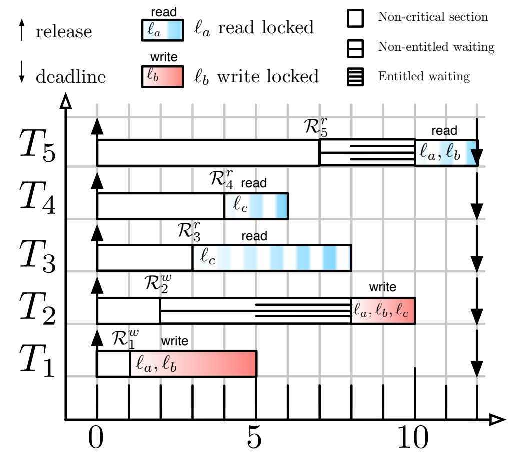
General rules. The first few rules of the R/W RNLP are common to both readers and writers and describe the necessary actions that must be taken when a job either issues a request or completes a critical section.

- **G1** When J_i issues $\mathcal{R}_i$ at time t, the timestamp of the request is recorded: $ts(\mathcal{R}_i) := t$.
- **G2** When $\mathcal{R}_i$ is satisfied, it is dequeued from either Q_a^r (if it is a read request) or Q_a^w (if it is a write request) for each $\ell_a \in \mathcal{D}_i$.
- **G3** When $\mathcal{R}_i$ completes, it unlocks all resources in $\mathcal{D}_i$.
- **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 (cont'd). At time $t = 8$, when $\mathcal{R}_3^r$ completes its critical section, $\mathcal{D}_3 = \{\ell_c\}$ is unlocked. This allows $\mathcal{R}_2^w$ to be satisfied (as explained later), and therefore $\mathcal{R}_2^w$ is dequeued from Q_a^w, Q_b^w, and Q_c^w.

The remaining read- and write-specific rules rely on the concept of *entitlement*, which we use to resolve the R/W ordering dilemma. Intuitively, a request becomes *entitled* once it is the next request to be satisfied (w.r.t. the resources

[1]Read sharing is reflexive and symmetric.



(a) Schedule in the running example.

time	queue states					
	Q_a^r	Q_a^w	Q_b^r	Q_b^w	Q_c^r	Q_c^w
$[0, 2)$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$[2, 7)$	$\emptyset$	$\{\mathcal{R}_2^w\}$	$\emptyset$	$\{\mathcal{R}_2^w\}$	$\emptyset$	$\{\mathcal{R}_2^w\}$
$[7, 8)$	$\emptyset$	$\{\mathcal{R}_2^w\}$	$\{\mathcal{R}_5^r\}$	$\{\mathcal{R}_2^w\}$	$\emptyset$	$\{\mathcal{R}_2^w\}$
$[8, 10)$	$\emptyset$	$\emptyset$	$\{\mathcal{R}_5^r\}$	$\emptyset$	$\emptyset$	$\emptyset$
$[10, 12]$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

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

Figure 3: Illustration of the running example.

for which it is waiting), and remains entitled until it is satisfied. While a request is entitled, it blocks all conflicting requests. Entitlement is defined differently for read and write requests. We begin with read requests, which are entitled if they are blocked only by satisfied (and not entitled) writes.

Def. 3. An unsatisfied read request $\mathcal{R}_i^r$ becomes *entitled* when there exists $\ell_a \in \mathcal{D}_i$ that is write locked, and for each resource $\ell_a \in \mathcal{D}_i$, $E(Q_a^w)$ is not entitled (see Def. 4).[2] (Note that $E(Q_a^w) = \emptyset$ could hold. In this case, we consider $E(Q_a^w)$ to be a "null" request that is not entitled.) $\mathcal{R}_i^r$ 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).

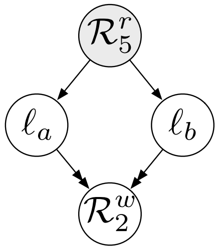
Example (cont'd). At time $t = 8$, $\mathcal{R}_5^r$ is blocked by $\mathcal{R}_2^w$, which holds ℓ_a, ℓ_b, and ℓ_c, as depicted in Fig. 4(a). By Def. 3, $\mathcal{R}_5^r$ becomes entitled at time $t = 8$ because ℓ_a and ℓ_b are write locked and $E(Q_a^w) = E(Q_b^w) = \emptyset$.

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

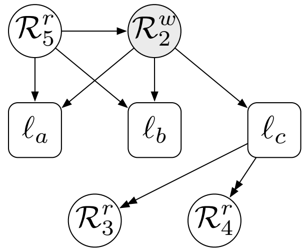
Def. 4. An unsatisfied write request $\mathcal{R}_i^w$ becomes *entitled* when for each $\ell_a \in \mathcal{D}_i$, $\mathcal{R}_i^w = E(Q_a^w)$, no read request in Q_a^r is entitled (see Def. 3)[3], and ℓ_a is not write locked. $\mathcal{R}_i^w$

[2] Entitlement is a property of a request, and Defs. 3 and 4 give conditions upon which a request becomes entitled in terms of the entitlement of other requests. Therefore, while Defs. 3 and 4 reference each other parenthetically to aid the reader, they are not in fact circularly defined.

[3]Sse footnote 2.



(a) Reader entitlement.



(b) Writer entitlement.

Figure 4: Illustrations of the wait-for graphs of entitled read and write requests. Inset (a) corresponds to $\mathcal{R}_5^r$ at time $t = 8$, and (b) corresponds to $\mathcal{R}_2^w$ at time $t = 7$ in Fig. 3. Note that in inset (b), $\mathcal{R}_5^r$ is blocked by at least one satisfied write request, and in (b) $\mathcal{R}_2^w$ is blocked by at least one satisfied write request.

remains entitled until it is satisfied.

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

Example (cont'd). At time $t = 7$, $\mathcal{R}_3^r$ holds ℓ_c, and blocks $\mathcal{R}_2^w$, which is waiting for ℓ_a, ℓ_b, and ℓ_c, as depicted in Fig. 4(b). Because $\mathcal{R}_2^w$ is the earliest timestamped writer waiting for any of the resources, and none is write locked, $\mathcal{R}_2^w$ becomes entitled. Note that, although $\mathcal{R}_2^w$ is entitled, it is still blocked. Prior to $t = 5$, $\mathcal{R}_2^w$ was not be entitled because ℓ_a and ℓ_b were write locked by $\mathcal{R}_1^w$.

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

Example (cont'd). At time $t \in [6, 8)$, $\mathcal{R}_2^w$ is blocked by $\mathcal{R}_3^r$, thus $B(\mathcal{R}_2^w, t) = \{\mathcal{R}_3^r\}$. Earlier, at time $t \in [5, 6)$, $\mathcal{R}_2^w$ is blocked by both $\mathcal{R}_3^r$ and $\mathcal{R}_4^r$, so $B(\mathcal{R}_2^w, t) = \{\mathcal{R}_3^r, \mathcal{R}_4^r\}$.

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

R1 When $\mathcal{R}_i^r$ is issued, for each $\ell_a \in \mathcal{D}_i$, $\mathcal{R}_i^r$ 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 (cont'd). At time $t = 3$, $\mathcal{R}_3^r$ is issued and it is satisfied immediately by Rule R1. $\mathcal{R}_3^r$ is allowed to "cut ahead" of $\mathcal{R}_2^w$ in this case because $\mathcal{R}_2^w$ is not entitled, and ℓ_c is unlocked. Further, at time $t = 10$, $\mathcal{R}_5^r$ is satisfied by Rule R2. This is because $\mathcal{R}_5^r$ is entitled, and $\mathcal{R}_2^w$ completed its critical section and unlocked ℓ_a and ℓ_b.

Writer rules. The writer rules parallel the reader rules.

W1 When $\mathcal{R}_i^w$ is issued, for each $\ell_a \in \mathcal{D}_i$, $\mathcal{R}_i^w$ 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 $B(\mathcal{R}_i^w, t) = \emptyset$.

Full example. At time $t = 1$, a write request $\mathcal{R}_1^w$ is issued for ℓ_a and ℓ_b, which is immediately satisfied (by Rule W1). At time $t = 2$, another write request, $\mathcal{R}_2^w$, is issued for ℓ_a, ℓ_b, and ℓ_c and is enqueued in Q_a^w, Q_b^w, and Q_c^w (by Rule W1). $\mathcal{R}_3^r$ is issued and satisfied immediately at time $t = 3$ by Rule R1, as previously described. Similarly, at time $t = 4$, $\mathcal{R}_4^r$ is issued and satisfied immediately (by Rule R1). Note that at time $t = 4$, both $\mathcal{R}_3^r$ and $\mathcal{R}_4^r$ 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 $\mathcal{R}_1^w$ completes at time $t = 5$, $\mathcal{R}_2^w$ becomes entitled. At time $t = 7$, $\mathcal{R}_5$ is issued for ℓ_b and ℓ_c, but it is not satisfied because $\mathcal{R}_2^w$ is entitled to both resources. After $\mathcal{R}_3^r$ completes at time $t = 8$, $\mathcal{R}_2^w$ is satisfied (by Rule W2). Finally, after $\mathcal{R}_2^w$ completes at time $t = 10$, $\mathcal{R}_5^r$ 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 resolve the R/W ordering dilemma, we have introduced the concept of entitled blocking. Intuitively, an entitled request is "next in line" with regard to its requested resources and only blocked by satisfied, but incomplete requests of the opposite kind.

B. Analysis

We now present blocking analysis for the R/W RNLP. Our analysis uses the following two corollaries, which follow from Lemmas 4 and 5, respectively, proved in Appendix A.

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

Example (cont'd). This corollary is demonstrated at time $t = 7$ in Fig. 5, when $\mathcal{R}_5^r$ is issued. Because $\mathcal{R}_2^w$ is entitled at that time, $\mathcal{R}_5^r$ is forced to block until after $\mathcal{R}_2^w$ completes, even though the resources it requested are available.

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

While Cor. 2 is not depicted in Fig. 5, it is similar Cor. 1. Next, 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 1. *A write request $\mathcal{R}_i^w$ experiences acquisition delay of at most L_{max}^r time units after becoming entitled.*

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

The following lemma is essential to show that transitive early-on-late blocking does not adversely affect the worst-case blocking bounds.

Lemma 2. *If $\mathcal{R}_i^w$ is the earliest timestamped write request among all incomplete write requests, then $\mathcal{R}_i^w$ is either satisfied or entitled.*

Proof: Suppose not. Then, by Def. 4, for some resource $\ell_a \in \mathcal{D}_i$, 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 Rule W1, (i) and (iii) are not possible since the write queues are timestamp ordered, and $\mathcal{R}_i^w$ is the earliest incomplete write. For (ii), assume $\mathcal{R}_x^r$ is entitled and $\ell_a \in \mathcal{D}_i \cap \mathcal{D}_x$. Then, by Def. 3, $\mathcal{R}_x^r$ is blocked by a satisfied write request $\mathcal{R}_h^w$. Recall that $\mathcal{R}_h^w$ must request all resources in the read sets of resources in $\mathcal{N}_h$. Further, ℓ_a must be in at least one of these read sets. Thus, $\ell_a \in \mathcal{D}_h \cap \mathcal{D}_i$, and $\mathcal{R}_h^w$ and $\mathcal{R}_i^w$ conflict. Thus, since $ts(\mathcal{R}_i^w) < ts(\mathcal{R}_h^w)$, $\mathcal{R}_h^w$ cannot be satisfied. ■

Theorem 1. *The worst-case acquisition delay of a read request $\mathcal{R}_i^r$ is at most $L_{max}^w + L_{max}^r$ time units.*

Proof: We first show that if $\mathcal{R}_i^r$ is issued at time t_i, then it must become entitled or satisfied by time $t_i + L_{max}^r$. Suppose not. Then, throughout the interval $[t_i, t_i + L_{max}^r)$, $\mathcal{R}_i^r$ is blocked by a non-empty set W of conflicting entitled write requests, for otherwise, $\mathcal{R}_i^r$ would become entitled (by Def. 3) or satisfied (by Rule R1). By Prop. P1 and Lemma 1, each write request $\mathcal{R}_x^w \in W$ will be satisfied by time $t_i + L_{max}^r$. Once all such write requests are satisfied, by Def. 3, $\mathcal{R}_i^r$ will become entitled or satisfied, a contradiction.

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

Theorem 2. *The worst-case acquisition delay of a write request $\mathcal{R}_i^w$ is at most $(m-1)(L_{max}^r + L_{max}^w)$ time units.*

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

The remaining possibility to consider is that $\mathcal{R}_x^w$ is entitled at t_i. In this case, by Def. 4, $\mathcal{R}_x^w$ is blocked by some read request $\mathcal{R}_h^r$. Thus, by Prop. P2, there are at most $m-2$ incomplete write requests with timestamps earlier than that of $\mathcal{R}_i^w$ at t_i. Reasoning as above, it follows that $\mathcal{R}_i^w$'s acquisition delay is at most $(m-2)(L_{max}^r + L_{max}^w) + L_{max}^r$ time units. (Note that the blocking of $\mathcal{R}_x^w$ due to $\mathcal{R}_h^r$ is accounted for in this reasoning by Lemma 1.) ■

For a spin-based lock, the worst-case acquisition delay for either reads or writes is the worst-case s-blocking (recall Def. 2). However, 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 $O(m)$ through analysis similar to single-resource spin-based mutex or reader-writer locks [3, 5].

In the remainder of this section, we briefly summarize four additional optimizations that can be incorporated into the R/W RNLP to improve average-case parallelism, and thus responsiveness in many cases. These optimizations do not affect the worst-case blocking bounds. We describe them independently for ease of exposition, but note that they can be combined in a real implementation. We note that the improved average-case responsiveness these optimizations provide result in larger safety margins, which are significant in safety-critical systems. While such systems must be proved correct, such proofs are built upon hardware models and system assumptions that may perhaps be incorrect, and thus larger safety margins are desirable in practice.

C. Requesting Fewer Resources

Requiring write requests to lock an expanded set of resources enabled us to establish Lemma 2. This lemma can instead be established by utilizing *placeholders*, which allow for increased parallelism. Specifically, we require a write request $\mathcal{R}_i^w$ to enqueue a *placeholder* $\mathcal{R}_i^p$ in the queues of all non-needed resources that we earlier required $\mathcal{R}_i^w$ to request in Sec. III-A. In this case, the R/W RNLP functions as previously described with the following exceptions. A placeholder is never entitled or satisfied. Instead, each placeholder $\mathcal{R}_i^p$ is removed from the write queue in which it is enqueued when $\mathcal{R}_i^w$ becomes entitled or satisfied. Therefore, until $\mathcal{R}_i^w$ becomes entitled, its associated placeholders prevent later-issued write requests from becoming entitled or satisfied, thereby ensuring that Lemma 2 is not violated.

Using placeholders, 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 [3, Chaps. 5,6].

D. R/W Mixing

Before we show how to relax Assumption 1 to allow jobs to issue *mixed* requests, we first extend our notation. We denote the set of resources that $\mathcal{R}_i$ needs read (write) access to as $\mathcal{N}_i^r$ ($\mathcal{N}_i^w$) and we let $\mathcal{N}_i = \mathcal{N}_i^r \cup \mathcal{N}_i^w$. If $\mathcal{N}_i^w = \emptyset$, then we say $\mathcal{R}_i$ is a read request, otherwise we say that $\mathcal{R}_i$ is a write request. With this notation, a mixed request is a write request $\mathcal{R}_i^w$ with $\mathcal{N}_i^r \neq \emptyset$ and $\mathcal{N}_i^w \neq \emptyset$. 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 $\mathcal{R}_i$, $\ell_a \in \mathcal{N}_i$, and $\ell_b \in \mathcal{N}_i^r$.[4]

The rules of the R/W RNLP support mixed requests 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, an entitled mixed request can be satisfied if all resources for which it requires read access are either unlocked *or* 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.

E. R-to-W Upgrading

We call a read request that can be upgraded to a write request, as previously described, an *upgradeable* request, which we denote as $\mathcal{R}_i^u$. Intuitively, we treat an upgradeable request as a write request that can optimistically execute read-only code while its 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 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 R/W RNLP, we treat $\mathcal{R}_i^u$ as two separate requests, a read request,[5] $\mathcal{R}_i^{u_r}$ and a write request $\mathcal{R}_i^{u_w}$, which can cancel each other if necessary.[6] When $\mathcal{R}_i^u$ is issued, $\mathcal{R}_i^{u_r}$ is enqueued as a read request and $\mathcal{R}_i^{u_w}$ is enqueued as a write request. If $\mathcal{R}_i^{u_w}$ is satisfied before $\mathcal{R}_i^{u_r}$, then $\mathcal{R}_i^{u_r}$ is canceled and removed from all read queues. If $\mathcal{R}_i^{u_r}$ is satisfied first, it executes its critical section, and upon completion or realization that upgrading is not necessary, $\mathcal{R}_i^{u_w}$ is canceled and removed from all write queues in which it is enqueued. If $\mathcal{R}_i^u$ must be upgraded, then when the read-only segment of its critical section completes, all resources are unlocked. Later, when $\mathcal{R}_i^{u_w}$ is satisfied, the job can execute the write segment of its critical section. Note that the state of any read objects may change between $\mathcal{R}_i^{u_r}$ completing and $\mathcal{R}_i^{u_w}$ being satisfied. Thus, $\mathcal{R}_i^{u_w}$ may need to re-read data. If this behavior is unacceptable for a given application, a write request should instead be issued for all resources that could be written.

F. Incremental locking

Next, we show how the R/W RNLP can be adapted to allow jobs to incrementally request resources they use within a critical section, as described earlier. We assume that it is known a priori the set of 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) [9].

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

Note that entitlement serves a similar purpose as priority ceilings [9], since it prevents later-issued requests from acquiring resources that may be incrementally requested.

IV. Evaluation

To evaluate the practicality of the R/W RNLP, we implemented the spin-based variant (without the optimizations just described), and conducted a *schedulability* study, in which we applied a *schedulability test* to tens of thousands of randomly generated task systems to determine the fraction of systems for which it could be shown that no deadlines are missed.

Implementation. We implemented the R/W RNLP in user-space on top of LITMUS$^{\mathrm{RT}}$ [7], a real-time extension of Linux. Our implementation was designed for a partitioned scheduler ($c = 1$); the partitioned earliest-deadline-first (EDF) scheduler was used in our evaluations. The implementation, for which pseudocode is available online [11],

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

[5]We assume the worst-case execution time of the read-only segment of the upgradeable request finishes in L_{max}^r time.

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

utilizes two locks, a mutex lock, and a phase-fair reader writer lock, as well as a novel wait-free technique. The phase-fair lock guards access to bitmasks, which store per-resource state, and the mutex lock protects access to all write queues. Importantly, in our implementation, read requests need only acquire read access to the phase-fair lock, and critical sections for this lock (both read and write) are very short—only a few instructions.

We evaluated our implementation on a 2.67Ghz quad-core Intel Core i7-920 processor. We measured the *overhead* of the lock and unlock procedures used in the implementation, where such overhead is defined to be the total procedure runtime minus any time spent busy waiting for other requests to complete. In total, we measured overheads for 18 task system configurations, similar to those presented in the schedulability study below. These task systems were chosen to ascertain how the implementation behaved under high contention, and under different ratios of read to write requests. Each task set was executed for two minutes. The largest median- and worst-case overheads observed across all task sets are reported in Tbl. I. These overheads are sufficiently small to demonstrate that the R/W RNLP can be practically implemented. Furthermore, read requests have smaller overheads than writes, which is desirable for a R/W locking protocol that is best used when reads are more common than writes.

Schedulability. Next, we present an evaluation of the R/W RNLP on the basis of schedulability. These experiments are intended to show the effects that blocking bounds have on schedulability, and do not include overheads. A full experimental evaluation incorporating overheads is beyond the scope of this paper and is deferred to future work.

We randomly generated sporadic task systems using a similar experimental design as previous studies (*e.g.*, [6]). We assume that tasks are partitioned onto $m = 4$ processors, and scheduled in EDF order.[7] We generated task systems with a total system utilization in $\{0.1, 0.2 \ldots, 4.0\}$. Per-task utilizations in a given task system were chosen to be *medium* or *heavy*, which correspond to uniformly distributed utilizations in the range $[0.1, 0.4]$ or $[0.5, 0.9]$, respectively. The periods of all tasks were chosen uniformly from either $[3, 33]\,\mathrm{ms}$ (*short*) or $[50, 250]\,\mathrm{ms}$ (*long*). All tasks were assumed to access shared resources, but only $\mathcal{P}^r \in \{50, 70, 90\}\%$ of the tasks issue read requests. Each read (write) request was configured to accesses $N^r \in \{1, 2, 4\}$ (resp., $N^w \in \{1, 2, 4\}$) of 50 resources. Read and write critical sections lengths for each job were exponentially distributed with a mean of either $10\,\mu\mathrm{s}$ (*small*) or $1000\,\mu\mathrm{s}$ (*long*).

For each generated task set, we evaluated schedulability using four different real-time locking protocols, OMLP mutex group locks [6], the RNLP [10], Phase Fair (PF) R/W group locks [5, 6], and the R/W RNLP presented herein.

[7]In many safety-critical embedded-systems domains, enabling a quad-core processor to be used would be a tremendous innovation.

Procedure	Median (μs)	Worst (μs)
read lock	0.106	0.168
read unlock	0.048	0.124
write lock	0.478	0.626
write unlock	0.129	0.215

Table I: Lock and unlock overheads for read and write requests. The worst-case overhead reported is the 99^{th} percentile, to filter the effects of interrupts and other spurious behavior.

Blocking bounds under each protocol were evaluated using fine-grained analysis similar to that in [3]. For the RNLP and the R/W RNLP, additional optimizations were also included, which are based on evaluating possible transitive blocking relationships.[8] In future work, we plan to explore linear-programming-based blocking analysis techniques, similar to those recently presented by Brandenburg [2].

While our experiments generated hundreds of schedulability graphs (available online [11]), here we present in Fig. 5 a small selection that depict relevant trends. In Fig. 5, the curves denoted NOLOCK depict schedulability assuming no resource requests, while the remaining curves depict schedulability using the locking protocol as labeled. The protocol with a curve closest to NOLOCK provides the best schedulability.

Obs. 1. In all observed cases, schedulability under the fine-grained locking protocols, the RNLP and the R/W RNLP, was no worse than schedulability using the corresponding coarse-grained locking protocols, the OMLP and phase-fair R/W locks, respectively.

This observation is supported by insets (a) and (b) of Fig. 5. In inset (a), the R/W RNLP is roughly the same as phase-fair R/W locks, while the RNLP significantly outperforms the OMLP. However, in many cases, such as in inset (b), the fine-grained RNLP and R/W RNLP offer improved schedulability over their coarse-grained counterparts.

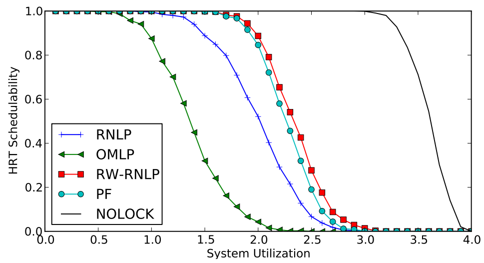
Obs. 2. For read-dominated workloads, *i.e.*, those with larger $\mathcal{P}^r$, phase-fair R/W locks, and the R/W RNLP perform comparatively better. The R/W RNLP performs comparatively better than phase-fair locks when N^r is small.

This observation is supported by inset (a) of Fig. 5, in which read critical section lengths are large and write critical section lengths are small, and 90% of tasks issue read requests. Additionally, in inset (b), in which $N^r = 1$, schedulability under the R/W RNLP is better than under phase-fair locks. Note that the gap between phase-fair locks and the R/W RNLP is smaller for larger N^r on account of write requests being forced to request unneeded resources.

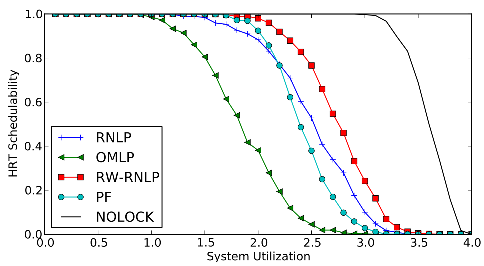
Obs. 3. In some cases in which there are a comparatively large number of write requests, the RNLP offers slightly improved schedulability over the R/W RNLP.

This observation is supported by inset (c) of Fig. 5, in

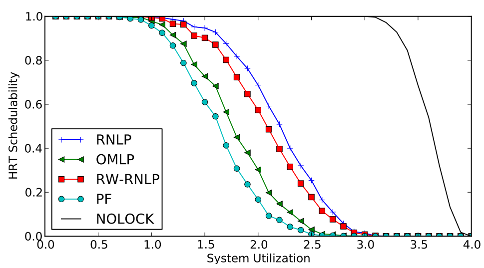
[8]Tighter analysis than that employed in these experiments is possible using an exponential-time algorithm. While this may be feasible for some task systems, it is too expensive in schedulability studies, which evaluate the schedulability of tens of thousands of task systems.



(a) $\mathcal{P}^r = 90\%$, $N^r = 2$, $N^w = 4$, large read and small write critical sections.



(b) $\mathcal{P}^r = 50\%$, $N^r = 1$, $N^w = 2$, large read and small write critical sections.



(c) $\mathcal{P}^r = 50\%$, $N^r = 2$, $N^w = 4$, large read and write critical sections.

Figure 5: Schedulability results.

which 50% of tasks issue write requests. Under the R/W RNLP, writer blocking is increased ($(m-1)(L^r_{max}+L^w_{max})$ instead of $(m-1)L_{max}$ for the RNLP) to allow for $O(1)$ reader blocking. When writes are comparatively common, the benefits of $O(1)$ reader blocking in some cases do not outweigh the cost of increased writer blocking, and thus the RNLP may outperform the R/W RNLP by a small margin.

These schedulability results, in conjunction with our measured overheads, demonstrate that fine-grained mutex and R/W locks are practically implementable, and offer improved schedulability over coarse-grained alternatives.

V. Conclusions

We have presented the R/W RNLP, which is the first fine-grained real-time multiprocessor locking protocol that supports reader/writer sharing. Having to support two different operations on resources—reads and writes—introduces considerable difficulty in designing a fine-grained reader/writer real-time locking protocol. The R/W RNLP resolves the R/W ordering dilemma using the concept of entitled waiting. The R/W RNLP also prevents transitive early-on-late blocking that would increase worst-case pi-blocking bounds.

We implemented the R/W RNLP and measured lock/unlock overheads, which were small. We also presented the first schedulability study of fine-grained locking under the RNLP and the R/W RNLP. These results suggest that these fine-grained locking protocols are useful in practice.

Acknowledgment: We thank Björn Brandenburg for insightful discussions, which helped identify some of the complexities that phase-fair behavior introduces to fine-grained locking protocols.

References

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

[2] B. Brandenburg. Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling. In *RTAS '13*.

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

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

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

[6] B. Brandenburg and J. Anderson. The OMLP family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Systems*, online first, July 2012.

[7] LITMUS$^{\mathrm{RT}}$ Project. http://www.litmus-rt.org/.

[8] A. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD thesis, MIT, Cambridge, MA, 1983.

[9] 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.

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

[11] B. Ward and J. Anderson. Multi-resource real-time reader/writer locks for multiprocessors. 2014. http://www.cs.unc.edu/~bcw/pubs/ipdps14long.pdf.

Appendix A: Entitlement 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 $\mathcal{R}_i$ if $\mathcal{R}_i$ becomes entitled (satisfied) as a result of I (*i.e.*, $\mathcal{R}_i$ is entitled (satisfied) after I but not before I).

Lemma 3. *The following properties of satisfaction and entitlement hold.*

- ***E1*** *If I satisfies $\mathcal{R}^r_i$, then I is either a read issuance or a write completion.*
- ***E2*** *If I satisfies $\mathcal{R}^w_i$, then I is either a write issuance, a read completion, or a write completion.*
- ***E3*** *If I satisfies $\mathcal{R}^r_i$ and I is the issuance of read request $\mathcal{R}^r_x$, then $\mathcal{R}^r_i = \mathcal{R}^r_x$.*
- ***E4*** *If I satisfies $\mathcal{R}^w_i$ and I is the issuance of write request $\mathcal{R}^w_x$, then $\mathcal{R}^w_i = \mathcal{R}^w_x$.*

E5 *If I satisfies $\mathcal{R}_i^w$ and I is the completion of a conflicting read request $\mathcal{R}_x^r$, then at time t_I^-, $\mathcal{R}_i^w$ is entitled, and $B(\mathcal{R}_i^w, t_I^-) = \{\mathcal{R}_x^r\}$.*

E6 *If I satisfies $\mathcal{R}_i^r$ and I is the completion of a conflicting write request $\mathcal{R}_x^w$, then at time t_I^-, $\mathcal{R}_i^r$ is entitled, and $B(\mathcal{R}_i^r, t_I^-) = \{\mathcal{R}_x^w\}$.*

E7 *If I satisfies $\mathcal{R}_i^w$ and I is the completion of a conflicting write request $\mathcal{R}_x^w$, then at time t_I^-, for each $\ell_a \in \mathcal{D}_i^w$, $\mathcal{R}_i^w = E(Q_a^w)$ and no read request in Q_a^r is entitled, and for each resource $\ell_a \in \mathcal{D}_i$, ℓ_a is either locked by $\mathcal{R}_x^w$, or unlocked.*

E8 *If I entitles $\mathcal{R}_i^r$, then I is a read issuance or a read completion.*

E9 *If I entitles $\mathcal{R}_i^w$, then I is a write issuance or a write completion.*

E10 *If $\mathcal{R}_i^w$ and $\mathcal{R}_x^r$ 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 $\mathcal{R}_i^r$ is waiting, and hence cannot cause $\mathcal{R}_i^r$ to become satisfied. On the other hand, if I is a read completion and $\mathcal{R}_i^r$ is not entitled prior to I, then by Rule R2, I cannot cause $\mathcal{R}_i^r$ to become satisfied. If I is a read completion and $\mathcal{R}_i^r$ is entitled (and hence blocked) prior to I, then $B(\mathcal{R}_i^r, t_I^-)$ contains at least one write request; I cannot cause this write request to complete, thus following I, $\mathcal{R}_i^r$ remains entitled (and hence blocked).

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

Prop. E3. If I is the issuance of read request $\mathcal{R}_i^r$, 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 $\mathcal{R}_i^r$ itself to become satisfied.

Prop. E4. Follows similarly to Prop. E3.

Prop. E5. By Rule W2, if I satisfies $\mathcal{R}_i^w$, then prior to I, $\mathcal{R}_i^w$ must have been entitled, and $\mathcal{R}_x^r$ must have been the only request that blocked $\mathcal{R}_i^w$.

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

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

Prop. E8. By Def. 3, if $\mathcal{R}_i^r$ 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, $\mathcal{R}_x^w$. Thus, by Def. 4, the following hold at time t_I^-: $\mathcal{R}_x^w$ is at the head of each write queue in which it is enqueued; no resource for which $\mathcal{R}_x^w$ is waiting is write locked; and $\mathcal{R}_x^w$ is not blocked by any entitled read request. Recall that entitled requests are, by definition, unsatisfied. Thus, $\mathcal{R}_x^w$ must be blocked by at least one *satisfied* read request at t_I^-. Now, if I is a write issuance, then $\mathcal{R}_x^w$ clearly remains entitled at t_I, and hence $\mathcal{R}_i^r$ 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 $\mathcal{R}_x^w$ to complete. Thus, as before, $\mathcal{R}_x^w$ remains entitled at t_I, and hence $\mathcal{R}_i^r$ 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 $\mathcal{R}_i^w$ is entitled, no conflicting request $\mathcal{R}_x$ can be satisfied before it, which implicitly bounds how long it remains entitled.

Lemma 4. *If a write request $\mathcal{R}_i^w$ is entitled before and after I and $\mathcal{R}_x \in B(\mathcal{R}_i^w, t_I)$, then $\mathcal{R}_x \in B(\mathcal{R}_i^w, t_I^-)$.*

Proof: Suppose not. Then the mentioned request $\mathcal{R}_x$ (read or write) is satisfied by I, and by the definition of $B(\mathcal{R}_i^w, t_I)$, $\mathcal{R}_x$ conflicts with $\mathcal{R}_i^w$.

Assume that $\mathcal{R}_x^r$ 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, $\mathcal{R}_x^r$ is issued at t_I; however, by Rule R1, I cannot then satisfy $\mathcal{R}_x^r$ because $\mathcal{R}_i^w$ is entitled. If I is a write completion, then by Prop. E6, $\mathcal{R}_x^r$ is entitled at t_I^-; however, by Prop. E10, this implies that $\mathcal{R}_i^w$ is not entitled at t_I^-, contradicting the lemma statement.

Now assume that $\mathcal{R}_x^w$ 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 $\mathcal{R}_i^w$ and $\mathcal{R}_i^w$ conflict and share some resource ℓ_c. Moreover, by Prop. E7, $\mathcal{R}_x^w = E(Q_c^w)$ holds at t_I^-. However, by Def. 4, this contradicts the assumption that $\mathcal{R}_i^w$ is entitled at t_I^-. ■

Similar to Lemma 4, we next show that once a read request $\mathcal{R}_i^r$ becomes entitled, no conflicting request can be satisfied before it.

Lemma 5. *If a read request $\mathcal{R}_i^r$ is entitled before and after I and $\mathcal{R}_x^w \in B(\mathcal{R}_i^w, t_I)$, then $\mathcal{R}_x^w \in B(\mathcal{R}_i^w, t_I^-)$.*

Proof: Suppose not. Then, the mentioned write request $\mathcal{R}_x^w$ is satisfied by I, and by the definition of $B(\mathcal{R}_i^w, t_I)$, $\mathcal{R}_x^w$ conflicts with $\mathcal{R}_i^w$. 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 $\mathcal{R}_x^w$ itself; however, by Rule W1, I cannot satisfy $\mathcal{R}_x^w$, because $\mathcal{R}_i^r$ is entitled prior to I. If I is a read (resp., write) completion, then by Prop. E5 (resp., Prop. E7), $\mathcal{R}_x^w$ is entitled at t_I^-; however, by Prop. E10, this contradicts the assumption that $\mathcal{R}_i^r$ is entitled at t_I^-. ■