Light Reading: Optimizing Reader/Writer Locking for Read-Dominant Real-Time Workloads

Catherine E. Nemitz
University of North Carolina at Chapel Hill, NC, USA

Shai Caspin
University of North Carolina at Chapel Hill, NC, USA

James H. Anderson
University of North Carolina at Chapel Hill, NC, USA

Bryan C. Ward
MIT Lincoln Laboratory, Lexington, MA, USA

Abstract

This paper is directed at reader/writer locking for read-dominant real-time workloads. It is shown that state-of-the-art real-time reader/writer locking protocols are subject to performance limitations when reads dominate, and that existing schedulability analysis fails to leverage the sparsity of writes in this case. A new reader/writer locking-protocol implementation and new inflation-free schedulability analysis are proposed to address these problems. Overhead evaluations of the new implementation show a decrease in overheads of up to 70% over previous implementations, leading to throughput for read operations increasing by up to 450%. Schedulability experiments are presented that show that the analysis results in schedulability improvements of up to 156.8% compared to the existing state-of-the-art approach.

2012 ACM Subject Classification Computer systems organization → Real-time system architecture; Computing methodologies → Shared memory algorithms

Keywords and phrases Reader/writer, real-time, synchronization, spinlock, RMR complexity

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2021.6

Supplementary Material *Software (ECRTS 2021 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.7.1.3`

Funding Work was supported by NSF grants CNS 1563845, CNS 1717589, CPS 1837337, CPS 2038855, and CPS 2038960, ARO grant W911NF-20-1-0237, and ONR grant N00014-20-1-2698. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGS-1650116. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This work was also supported by a Dissertation Completion Fellowship from the UNC Graduate School.

Acknowledgements DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited. This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering. © 2021 Massachusetts Institute of Technology. Delivered to the U.S. Government with Unlimited Rights, as defined in DFARS Part 252.227-7013 or 7014 (Feb 2014). Notwithstanding any copyright notice, U.S. Government rights in this work are defined by DFARS 252.227-7013 or DFARS 252.227-7014 as detailed above. Use of this work other than as specifically authorized by the U.S. Government may violate any copyrights that exist in this work.

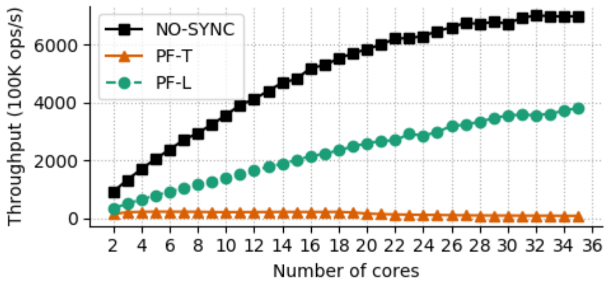


Figure 1 Throughput for red-black tree lookups. This read-only scenario is representative of use cases where writes can occur but are so infrequent that over long intervals, only reads occur.

1 Introduction

In an ongoing project, our research group has been investigating real-time use cases where shared resources exist that are much more frequently read than written. The importance of such read-dominant use cases has been well documented by McKenney [24], who devised a non-blocking synchronization solution called *Read-Copy-Update (RCU)* to support resource sharing. As explained later, RCU can be problematic in real-time systems, so for our use cases, a better option is *reader/writer locks*, a now-standard synchronization solution first proposed five decades ago [16]. A reader/writer lock extends a mutex lock by distinguishing between *read accesses* (non-modifying) and *write accesses* (potentially modifying) and seeks to allow reads to execute concurrently with one another while supporting exclusive writing.

At the outset of our project, it was our belief that reader/writer locking was a solved problem for real-time systems. This belief was rooted in the existence of reader/writer locking protocols that are asymptotically optimal with respect to blocking times [12], and the apparent ease with which such blocking times can be factored into schedulability analysis [3]. In delving further, however, we found this belief to be wrong. In particular, for read-dominant use cases, we found major deficiencies with respect to both state-of-the-art real-time reader/writer locking protocols and the schedulability analysis needed to apply them.

These experiences motivated this paper, which is directed at the goal of efficiently supporting read-dominant real-time workloads. Our contributions towards this goal include a new reader/writer locking-protocol implementation and schedulability analysis. We expound on these contributions below, after first elaborating on the deficiencies noted above.

Surprising performance limitations in existing reader/writer locks. In the real-time literature, Brandenburg and Anderson presented a category of reader/writer locks called *phase-fair locks* over a deacade ago [12] and established the optimality of such locks under common definitions of *priority-inversion blocking (pi-blocking)*. To our knowledge, phase-fair locks stand as the state-of-the-art for spin-based (our focus) real-time reader/writer locking.

In our work on read-dominant workloads, we employed a *phase-fair ticket lock (PF-T)*, one of Brandenburg and Anderson's proposed phase-fair variants [11]. In experiments involving PF-Ts, we observed perplexing behavior, shown in Fig. 1: throughput did not scale beyond four cores for a purely read-only workload. This was surprising as the phase-fair lock logic should allow all reads to execute without ever blocking. So why then did the PF-T not scale to match the case of having no synchronization at all (NO-SYNC)?

The answer relates to the overhead of the PF-T's lock/unlock logic. In particular, every lock and unlock call updates the shared lock state. Even under a read-only workload, these updates invalidate cached lock state on other cores. Contention for this shared lock state incurs significant overhead that severely hampers throughput. Additionally, many shared-state updates require the use of an atomic instruction.

Analytically leveraging the sparsity of writes. In examining existing schedulability analysis for phase-fair locks [11], we found that they do not exploit the sparsity of writes in read-mostly workloads. In particular, reads are pessimistically assumed to always incur some write blocking. Thus, even if lock performance close to NO-SYNC could be achieved, such improvements would be lost *analytically* in checking schedulability. In recent years, holistic, *inflation-free* blocking analysis has been developed for mutex locks that limits over-estimates of blocking. However, such analysis has not been extended to apply to reader/writer locks.

Contributions. The contributions of this paper are four-fold. First, in Sec. 3, we present the spin-based *phase-fair with light reading ticket lock (PF-L)*, which is optimized for read-dominant workloads. The PF-L achieves low read overhead by eliminating shared lock state between readers (at the expense of forcing write requests to check additional state) and by eliminating atomic instructions from read lock/unlock logic. It also enables *sequences* of reads to access cached lock state exclusively if they are uninterrupted by writes.

Second, in Sec. 4, we present an experimental evaluation of the PF-L compared to other alternatives on the basis of throughput and locking overheads. Across all of our experiments, the PF-L enabled throughput increases over the state-of-the-art PF-T in the range 2–450%, and overhead reductions for reading in the range 40–73% for read-dominant workloads.

Third, in Sec. 5, we extend prior inflation-free blocking analysis proposed for mutex locks [9] to apply to reader/writer locks. This analysis involves modifying an integer linear program (ILP), as the introduction of reads requires applying numerous additional constraints.

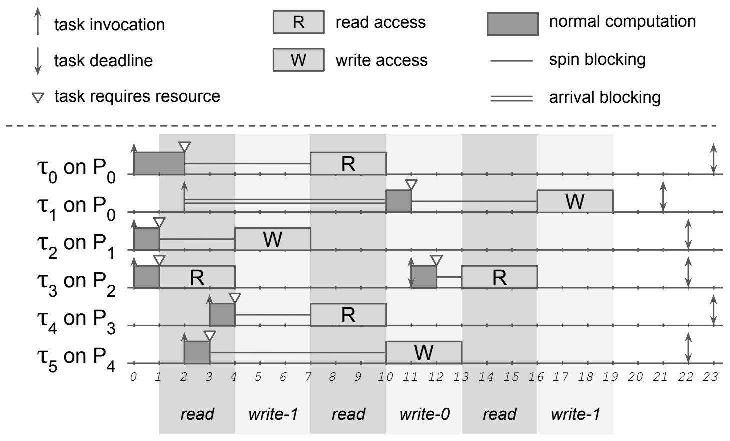
Fourth, in Sec. 6, we present the results of a schedulability study that we conducted to compare our new PF-L implementation and inflation-free reader/writer blocking analysis to prior alternatives. In this study, our new analysis improved schedulability by up to 159% compared to previous state-of-the-art methods.

2 Background

In this section, we present our assumed models and relevant background and related work.

Task model. We consider a sporadic task system Γ comprised of n constrained-deadline tasks scheduled by the Partitioned Earliest-Deadline-First (P-EDF) scheduler on a multiprocessor platform with m cores. (We assume familiarity with the sporadic model and P-EDF.) An arbitrary task is denoted τ_i. When conducting analysis, the partition (core) under consideration is denoted P^*, and an arbitrary partition is denoted P_k.

Resource model. We assume a set of n_r shared resources, with an arbitrary resource denoted ℓ_q. A job of a task can issue a request for only one resource at a time (no nesting). Each request is either a *read request* (which may execute concurrently with other reads) or a *write request* (which must be exclusive). We use $\mathcal{R}_i^r$, $\mathcal{R}_i^w$, and $\mathcal{R}_i$ to denote a read, write, or arbitrary (read or write) request, respectively, issued by a job of τ_i. Once a request $\mathcal{R}_i$ for a resource ℓ_q has been granted, $\mathcal{R}_i$ is *satisfied*, and the issuing job *holds* ℓ_q until $\mathcal{R}_i$ *completes*.



time	read queue	write queue
[0, 1)	-	-
[1, 2)	-	$\mathcal{R}_2$
[2, 3)	$\mathcal{R}_0$	$\mathcal{R}_2$
[3, 4)	$\mathcal{R}_0$	$\mathcal{R}_2$, $\mathcal{R}_5$
[4, 7)	$\mathcal{R}_0$, $\mathcal{R}_4$	$\mathcal{R}_2$, $\mathcal{R}_5$
[7, 10)	-	$\mathcal{R}_5$
[10, 11)	-	-
[11, 12)	-	$\mathcal{R}_1$
[12, 13)	$\mathcal{R}_3$	$\mathcal{R}_1$
[13, 16)	-	$\mathcal{R}_1$
[16, 21)	-	-

Figure 2 Phase-fair request satisfaction.

Phase-fair locks. Phase-fair (PF) reader/writer (RW) locks were proposed to improve blocking bounds in real-time systems [12]. Some prior approaches to RW locking can starve read (resp., write) requests by prioritizing writes (resp., reads) over them [16, 25]. PF locks instead employ alternating read and write phases. Assuming non-preemptive spin-based locking (as we do here), the orchestration of these phases is defined by four rules [12]:

PF1 reader phases and writer phases alternate;
PF2 writers are subject to FIFO ordering, but only with regard to other writers;
PF3 at the start of each reader phase, all currently unsatisfied reads are satisfied (exactly one write request is satisfied at the start of a writer phase); and
PF4 during a reader phase, newly issued read requests are satisfied only if there are no unsatisfied write requests pending.

▶ **Example 1.** Consider the six tasks depicted in Fig. 2. For simplicity, we assume that each task requires access to the same resource; the type of access is indicated for each request. At time $t = 1$, the first job of τ_3 issues a read request, $\mathcal{R}_3^r$, which is satisfied immediately. Just after this, at $t = 1 + \epsilon$, τ_2 issues a write request $\mathcal{R}_2^w$, which must wait (Rule PF1). At $t = 2$ and $t = 4$, τ_0 and τ_4 issue read requests $\mathcal{R}_0^r$ and $\mathcal{R}_4^r$, respectively, which also must wait (Rule PF4). At $t = 3$, τ_5 issues a write request, which waits; it will not be satisfied before the write request issued by τ_2 (Rule PF2). When $\mathcal{R}_3^r$ completes, $\mathcal{R}_2^w$ is satisfied (Rule PF1). When $\mathcal{R}_2^w$ completes, both $\mathcal{R}_0^r$ and $\mathcal{R}_4^r$ are satisfied (Rule PF3). Read and write phases continue to alternate, as shown. Note that the job of τ_1 released at $t = 2$ does not preempt the currently executing job of τ_0 on P_0 because the latter is executing non-preemptively.

Different implementation strategies can be applied to realize the phase-fair rules. Brandenburg and Anderson presented several implementations, including a ticket-lock-based implementation (PF-T), a more compact version for embedded systems (PF-C), and a queue-based implementation with $O(1)$ RMR time complexity (PF-Q) [12]; as discussed more fully later, RMR time complexity counts only operations that entail an interconnect traversal to access memory. An alternative $O(1)$ implementation has been given by Bhatt and Jayanti [8].

Blocking analysis. When checking schedulability, locking delays must be accounted for. The simplest approach involves *inflating* execution times by *per-request* worst-case blocking bounds. This approach is safe but pessimistic, as the worst case may not always occur.

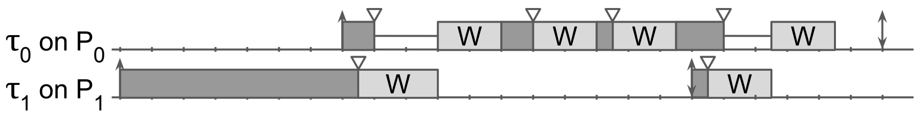


Figure 3 Example of blocking between two tasks.

Recently, *holistic, inflation-free* analysis, which accounts for blocking over an analysis interval, has been proposed for mutexes [9, 27, 30]. Such analysis seeks to avoid over-estimating locking delays.

▶ **Example 2.** Fig. 3 depicts a schedule of two tasks, τ_0 and τ_1, that write a common resource. Note that each request of τ_0 can potentially be blocked by one request of τ_1. Under conventional inflation-based blocking analysis, the execution time of τ_0 would thus be inflated by the cost of four requests. However, only two requests may be issued by τ_1 during one job of τ_0. Holistic analysis leverages such knowledge to more tightly bound total blocking, and an inflation-free approach accounts for this blocking without inflating task execution times.

Related work. As mentioned in Sec. 1, RCU, like our PF-L implementation, was designed for read-dominant workloads. However, RCU is not linearizable [19]. Furthermore, RCU efficiently executes reads by requiring writes to copy shared-object state and this copying results in a need for garbage collection. To our knowledge, no schedulability analysis exists for RCU, in part due to the non-deterministic behavior of garbage collection.

In addition to the RW locks already mentioned, FIFO-based and reader-preference locks have been developed for higher throughput [23], and reader-preference, writer-preference, and no-preference RW locks with $O(1)$ RMR time complexity (see Sec. 3) have been presented [7].

Of the PF variants by Brandenburg and Anderson mentioned earlier, the two ticket-lock-based approaches, the PF-T and PF-C, have $O(m)$ RMR time complexity, while the queue-based PF-Q has $O(1)$ RMR time complexity [12]. However, when measuring worst-case overheads, the sub-optimal PF-T outperforms the asymptotically optimal PF-Q [12] due to the lower frequency of atomic operations. Similarly, the $O(1)$ implementation of Bhatt and Jayanti [7] also includes multiple atomic instructions within the entry and exit sections of both reads and writes. We therefore focus our experiments to compare with PF-Ts.

Brandenburg and Anderson also developed suspension-based phase-fair implementations for clustered-scheduled systems [13].[1] Finally, Ward and Anderson applied phase-fair reasoning to support nested reader/writer locking [28].

3 The PF-L: A New Phase-Fair Lock with Light Reading

Our proposed PF-L is shown in Alg. 1. In this section, we described its motivation, data structures and how its code works, and analyze its RMR time complexity.

PF-L motivation. Recall from Fig. 1 that PF-T fails to scale for a read-only workload. We conjectured this was due to overheads, particularly cache-line bouncing and interconnect traffic triggered by updates to shared lock state. Therefore, we designed the PF-L to isolate, with respect to caches, lock state where possible, especially among reads on different cores.

[1] Clustered scheduling generalizes partitioned and global scheduling.

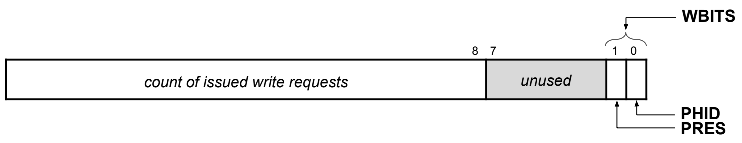


Figure 4 Usage of *win*.

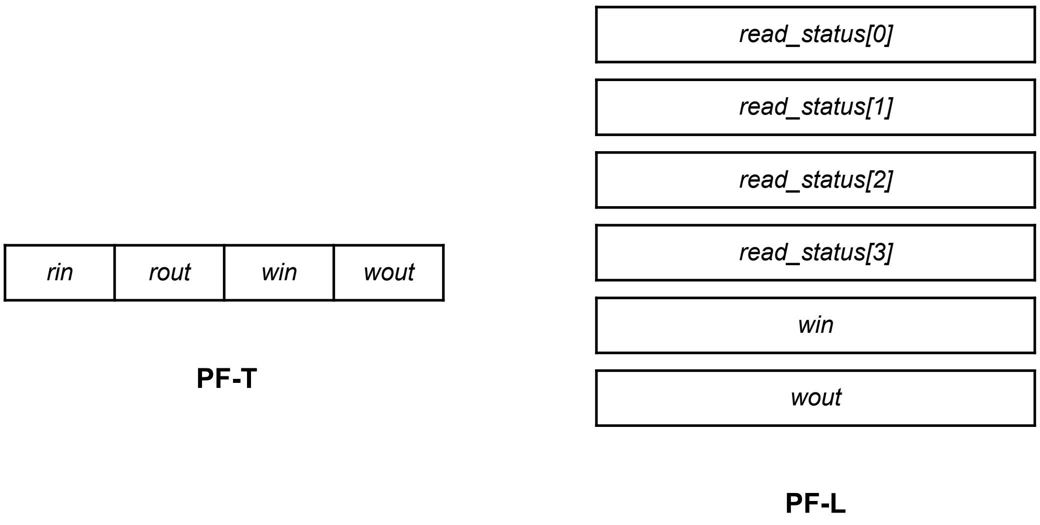


Figure 5 Distribution of locking protocol data structures across cache lines.

Data structures. Like the PF-T, the PF-L uses variables to count the number of write requests "in" and "out" (*i.e.*, the number issued and completed), *win* and *wout*, respectively. In the lowest-order byte of *win*, the PF-L maintains two bits indicating if a write request is present and the current write-phase – each write request is satisfied during either a 0-phase or a 1-phase. (The alternation of these phases prevents a race condition in which read requests could otherwise fail to distinguish the end of one write phase from the waiting of the subsequent active write request.) `WBITS` refers to these two bits, with `PRES` being the writer-present bit and `PHID` the write-phase bit, as illustrated in Fig. 4. The PF-T uses two global counters for the number of read reqests issued (*rin*) and the number of read requests completed (*rout*). In contrast, the PF-L uses a per-core variable, *read_status*, to maintain the status of any read requests. This difference is illustrated for a four-core system in Fig. 5, in which all of the variables for the PF-T are stored on the same cache line and each variable for the PF-L is allocated a separate cache line. Even if *rin* and *rout* were separated in the PF-T, all read requests would require updating those same locations. Instead, in the PF-L, a read request only updates *read_status* for its processor, avoiding conflicts with other read requests. This per-core definition enables isolating the variables to reduce cache interference (described in depth below). Coordination of read and write phases is achieved by requests updating and reading these variables. As such, this lock state is essential to ensuring linearizability [19].

Code description. We explain the code in Alg. 1 by walking through part of the example illustrated in Fig. 2. We describe the execution of the code at several time instants.

Algorithm 1 Phase-Fair with Light Reading (PF-L).

```
 1: type $res\_state$: record // all aligned on different cache lines
 2:     $read\_status$: array of unsigned integer, each initially COMPLETED      ▷ Cache aligned
 3:     $win$, $wout$: unsigned integer, initially 0

 4: constant
 5:     WINC 0x100  // writer increment value
 6:     WBITS 0x3  // writer bits in $win$
 7:     PRES 0x2  // writer-present bit
 8:     PHID 0x1  // write-phase bit
 9:     PRESENT 0x3  // reader present indicator
10:     COMPLETED 0x4  // reader completed indicator

11: procedure READ_LOCK($\ell$: ptr to $res\_state$, $k$: core index)
12:     var $w$: unsigned int
13:     $\ell \to read\_status[k]$ := PRESENT
14:     $w := \ell \to win$ & WBITS
15:     $\ell \to read\_status[k] := w$ & PHID      ▷ To wait on write phase ($w$ & PHID), if active
16:     await ($w$ & PRES $= 0$) or ($w \neq$ ($\ell \to win$ & WBITS))      ▷ Satisfied

17: procedure READ_UNLOCK($\ell$: ptr to $res\_state$, $k$: core index)
18:     $\ell \to read\_status[k]$ := COMPLETED

19: procedure WRITE_LOCK($\ell$: ptr to $res\_state$)
20:     var $wticket$, $read\_waiting$: unsigned int
21:     $wticket$ := fetch&add($\ell \to win$, WINC) and ¬WBITS      ▷ In write queue
22:     await ($wticket = \ell \to wout$)      ▷ Head of write queue
23:     fetch&xor($\ell \to win, 0x3$)      ▷ Marked present and new phase for reads to see
24:     $read\_waiting := \ell \to win$ & PHID
25:     for $k$ in core numbers do
26:         await ($read\_status[k] = read\_waiting$) or ($read\_status[k]$ = COMPLETED)

27: procedure WRITE_UNLOCK($\ell$: ptr to $res\_state$)
28:     fetch&and($\ell \to win$, 0xFFFFFF01)      ▷ Clear PRES, but keep PHID
29:     $\ell \to wout := \ell \to wout +$ WINC
```

Time $t = 1$. When $\mathcal{R}_3^r$ is issued on P_2, it first marks $read\_status[2]$ = PRESENT (Line 13). Then it reads the value in win (Line 14). This is the first request in the system, so $w = 0$. Now, $read\_status[2] = 0$, indicating that $\mathcal{R}_3^r$ would wait for a satisfied write request in Phase 0, if there is one, but none exists, so $\mathcal{R}_3^r$ is satisfied immediately (Line 16).

Just as $\mathcal{R}_3^r$ finishes executing READ_LOCK, $\mathcal{R}_2^w$ begins executing WRITE_LOCK. $\mathcal{R}_2^w$ increments win (Line 21), storing $wticket = 0$ and waits for $wticket = wout$ (Line 22). This serves as a ticket lock to ensure at most one write request is executing any of the following lines of WRITE_LOCK. Next, $\mathcal{R}_2^w$ sets the writer-present bit and flips the write-phase bit (Line 23), resulting in the last two bits of win holding 0b11. This is how the presence and phase of an active write request is shared with read requests. $\mathcal{R}_2^w$ then computes that a read waiting for the completion of its write phase would display a $read\_status$ value of 1. $\mathcal{R}_2^w$ next checks for active read requests on each core. For P_0 and P_1, it reads the $read\_status$ as COMPLETED and proceeds. However, for P_2, $\mathcal{R}_2^w$ reads $read\_status[2] = 0$. Thus, the read request on P_2 is not waiting for $\mathcal{R}_2^w$ but is satisfied; $\mathcal{R}_2^w$ waits for this request to complete.

Time $t = 2$. As illustrated in Fig. 2, while $\mathcal{R}_2^w$ waits, $\mathcal{R}_0^r$ is issued. At $t = 2$, the resource is in a read phase, but the waiting write request requires $\mathcal{R}_0^r$ to wait until the subsequent read phase (by Rule PF4). This is accomplished in READ_LOCK as follows. $\mathcal{R}_0^r$ sets $read\_status[0]$=PRESENT, stores $w = 3$, and sets $read\_status[0] = 1$. It then awaits a change in the WBITS of win, which will not occur until $\mathcal{R}_2^w$ completes. Note that from the perspective of $\mathcal{R}_2^w$, P_0 was already checked for active read requests, but the newly issued read request will safely wait based on the check of win, so no additional checks are required.

Time $t = 4$. Once $\mathcal{R}_3^r$ completes, and $\mathcal{R}_2^w$ becomes satisfied by the phase-fair rules. We now illustrate how that is accomplished in the PF-L. When $\mathcal{R}_3^r$ completes, it marks *read_status*[2] = `COMPLETED` (Line 18). Then $\mathcal{R}_2^w$ sees *read_status*[2] = `COMPLETED` and resumes checking cores. Next, $\mathcal{R}_2^w$ checks P_3 and sees *read_status*[3] = `PRESENT`, as $\mathcal{R}_4^r$ has just been issued. However, like $\mathcal{R}_0^r$, $\mathcal{R}_4^r$ soon sets *read_status*[3] = 1, indicating that the read request on P_3 is waiting for the execution of a write Phase 1 ($\mathcal{R}_2^w$'s write phase). $\mathcal{R}_2^w$ proceeds, reads *read_status*[4] = `COMPLETED`, and becomes satisfied.

Time $t = 7$. Once $\mathcal{R}_2^w$ completes, it clears the writer-present bit (Line 28); the last two bits of *win* subsequently hold 0b01, indicating that there is not a writer present and that the prior write phase was Phase 1. The waiting read requests, $\mathcal{R}_0^r$ and $\mathcal{R}_4^r$, observe this change and are satisfied immediately. Next $\mathcal{R}_2^w$ increments *wout* (Line 29), prompting $\mathcal{R}_5^w$ to execute the remaining logic of WRITE_LOCK.

RMR time complexity. The *remote memory references (RMR)* time-complexity measure was proposed in work on spin-based synchronization algorithms [33]. Under this measure, only operations that generate an interconnect traversal are counted; other operations are ignored. In applying this measure, architectural details are dealt with somewhat abstractly. In this work, we use a refined notion of RMR time complexity that incorporates such details.

Specifically, we assume a *write-back*, *write-invalidate* cache coherence protocol [17], which is consistent with many commodity processors (*e.g.*, x86 Intel and AMD processors). Abstractly, a write-back cache is one in which a memory write is cached and not written until later necessary (*e.g.*, due to a cache eviction). In a write-invalidate cache, when a memory write occurs, if that address is cached on a remote core, it is marked as *invalid* and subsequent accesses must be re-read. Any communication among caches is performed over an interconnect that all caches *snoop* or listen upon for any events that require updating their state. This interconnect introduces latency into cache and memory operations. We refer the reader to [18] for further discussion, but highlight the two most salient properties of such caches that influence the PF-L's design:

C1 When a cache block is written it becomes *write hot*. Any subsequent core-local reads or writes of that block do not generate interconnect traffic while the block is write hot. The block stays write hot until it is evicted, or read or written by another core.

C2 A cache block that is read that is not write hot becomes *read hot*. Any subsequent core-local reads of that block do not generate interconnect traffic while the block is read hot. The block stays read hot until it is evicted or modified by any core.

Given this model, we define a *local memory reference (LMR)* to be one in which no interconnect traversal is generated from the L1 data cache. For simplicity, we assume that atomic operations generate interconnect traffic, and are therefore not LMRs. Conversely, *remote memory references (RMRs)* are ones that are not local.

When analyzing RMR time complexity, we assume there are no *conflict misses*, *i.e.*, that there is sufficient cache space for all lock state to be cached concurrently. Furthermore, we assume cached lock state persists both during and between critical sections. Finally, we assume there are no cache evictions due to preemptions or migrations, as such costs are typically accounted for through separate analyses [6]. While in practice these assumptions may not always hold, they enable analysis of RMRs inherent to the protocol over a sequence of lock invocations, rather than on a per-invocation basis. This is relevant in cases where there is high lock contention, and potentially many requests to the same lock by one task.

RMR time complexity of the PF-L. Assuming the cache behavior defined above, the PF-L has $O(1)$ *amortized* RMR time complexity for an arbitrary sequence of r consecutive read requests on the same core uninterrupted by a write request, instead of $\Omega(r)$ as in all prior phase-fair approaches. Towards establishing this, we define a *read interval* to be an interval $[t, t')$ in which there is no pending or completed write request. Note that read requests from any and all cores may be issued and satisfied during a read interval. Now consider a read interval $[t, t')$ and a sequence of read requests $\mathcal{R}_1^r, \ldots, \mathcal{R}_r^r$ on core P^* that are issued after t and completed before t'. We make no assumption about the initial cache state at time t, and therefore at Line 16, $\mathcal{R}_1^r$ incurs an RMR to cache *win*. This leaves *win* read hot and *read_status*$[P^*]$ write hot on P^* when $\mathcal{R}_1^r$ completes. $\mathcal{R}_1^r$ therefore incurs $O(1)$ RMRs.

Continuing inductively, we show that each subsequent read request $\mathcal{R}_j^r$ where $j \in \{2, \ldots, r\}$, incurs no RMRs, and leaves the cache in the same state. First, observe that *win* is only modified by write requests (Lines 21, 23, and 28), which by definition do not occur in a read interval. Therefore, *win* will not be invalidated by $\mathcal{R}_j^r$, and will remain read hot (C2). Thus, any access to *win* by $\mathcal{R}_j^r$ will be an LMR.

Next, observe that *read_status*$[P^*]$ is **(i)** only modified by read requests on P^* (Lines 13, 15, and 18), and **(ii)** only read by write requests, which by definition do not occur in the read interval. Thus, the accesses to *read_status*$[P^*]$ are writes to a write-hot block, which are LMRs, and leave *read_status*$[P^*]$ write hot (C1). Taken together, this reasoning inductively proves $O(1)$ amortized RMR time complexity as claimed.

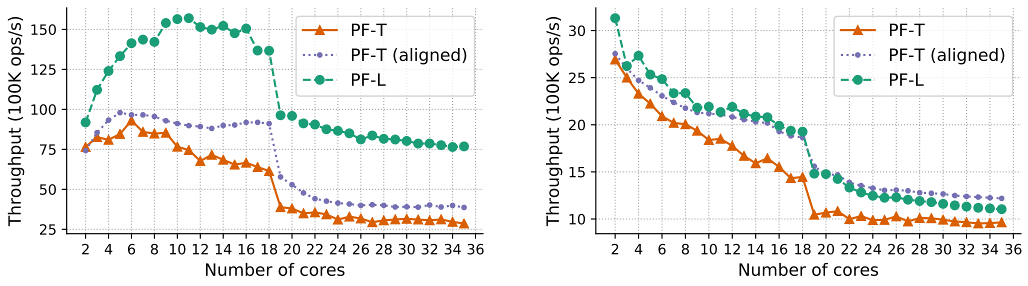
We note that, in the presence of write requests, reads in the PF-L have $O(m)$ RMR time complexity. In particular, before a read request is satisfied, it spins on *win*, which may be updated by at most $m - 2$ other newly issued write requests. Write requests in the PF-L clearly have $O(m)$ RMR time complexity. The spinning at Line 22 generates $O(m)$ RMRs (like any ticket lock), as does the FOR loop at Lines 25-26.

4 Evaluation of the PF-L

We empirically compared the PF-L to Brandenburg's PF-T implementation [11]. The results of this comparison include throughput graphs, including Fig. 1, as well as overhead data measured as a function of varying workloads. We conducted all experiments on a two-socket, 18-cores-per-socket x86 machine running the Linux 4.9.30 LITMUS$^{\text{RT}}$ kernel [2], with two Intel Xeon E5-2699 v3 CPUs @ 2.30 GHz, 128 GB of RAM, and three levels of cache: per-core 32 KB L1 data and instruction caches, 256 KB L2 caches shared by pairs of cores, and 46,080 KB L3 caches shared by all cores on the same socket. We performed each evaluation on $m \in \{2, \ldots, 36\}$ cores and two sockets. For $m \leq 18$, only one socket was used.

Recall that in the PF-L all lock-status variables are aligned to be cached on different lines. This allows each *read_status* variable to exist in a core-local L1 cache and never be invalidated by readers on other cores. Brandenburg's PF-T variables are all packed into a single cache line by design to minimize cache-line reloading costs [11]. All subsequent references to the PF-T are to Brandenburg's original implementation unless otherwise stated.

In conducting the following experiments, the contents of the cache were not protected. However, these experiments were conducted in isolation, so the cache behavior can be entirely attributed to the experiments. We did not conduct experiments in which another workload was designed to evict cache lines, as our focus was on capturing the overhead of cache evictions inherent to the execution of the protocol itself. There is prior work on protecting caches lines in real-time systems [4, 14, 15, 20, 21, 22, 29, 31, 32, 34], and one of these approaches could be applied to ensure competing workloads in a system do not evict the data structures of the locking protocol from the cache.



(a) 5% inserts, 95% lookups

(b) 50% inserts, 50% lookups

Figure 6 Red-black tree throughput.

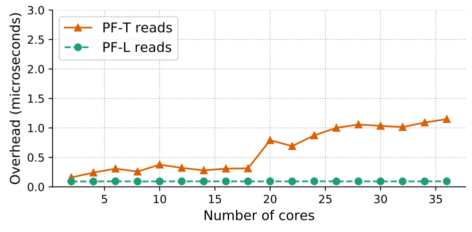
Throughput. We evaluated throughput using a realistic workload of lookups (reads) and inserts (writes) in a shared red-black tree, as this was the motivating use case that inspired this paper. Throughput was measured holistically to include locking overheads, blocking, and varying critical-section lengths based on whether an operation was a read or write. All operations were partitioned evenly across cores executing a single task per core. For each experiment, we averaged the throughput for ten unique random trees, each with a million nodes. The results for an *all-read* workload are shown in Fig. 1, which includes a plot for the same experimental setup with no synchronization. Fig. 6 presents throughput trends for a *read-dominant* workload and a workload with *evenly distributed* reads and writes.

▶ **Observation 3.** *The PF-L exhibited linear scaling with increasing core counts for an all-read workload.*

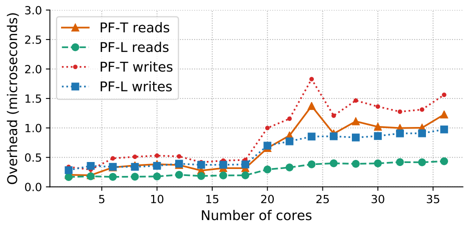
Fig. 1 highlights the pitfalls of the PF-T for an all-read workload. In comparison, throughput under the PF-L scaled linearly with the core count as the *read_status* variable was maintained write hot in the L1 data cache. Cache behavior enabled better scaling on one-socket for read-dominant workloads, as shown in Fig. 6a. On two sockets, throughput decreased with higher core counts due to more expensive interconnect operations. For more balanced workloads of reads and writes, as shown in Fig. 6b, throughput did not increase for either the PF-L or PF-T. This is because, with both reads and writes present, the RMR complexity for all requests in the PF-L is $O(m)$, as shown earlier. However, throughput was still higher by up to 25% than for the PF-T due to overheads.

After observing the benefits of cache-aligned variables in the PF-L, we tested aligning each PF-T variable in its own cache line. We discovered this version outperformed the original PF-T– a useful contribution in its own right. Throughput for the cache-aligned PF-T is also shown in Fig. 6; the cache-aligned PF-T actually performed similarly to the PF-L in the evenly distributed workload of reads and writes on high core counts.

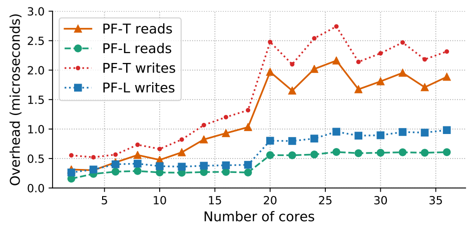
Overheads. In measuring overheads, it is necessary to distinguish time spent in operations inherent to the algorithm (overheads) from those incurred while spinning (blocking). For overhead-measurement purposes only, we instrumented both the PF-T and the PF-L to measure overheads and blocking separately. We recorded blocking and overhead times for 100,000 lock and unlock calls across an increasing number of cores. To simulate high contention and record worst-case overheads, critical sections were empty. All figures present the 99^{th} percentile observed overheads to filter outliers due to interrupts and other jitter due to userspace timing. Fig. 7 shows overhead trends for several different workloads. Overheads were measured separately for reads and writes, each including both lock and unlock costs.



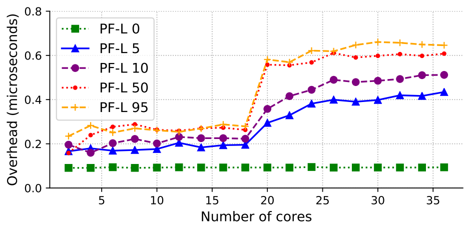
(a) All reads



(b) 5% writes



(c) 50% writes



(d) the PF-L with varying write %

Figure 7 Total overheads for lock and unlock operations.

▶ **Observation 4.** *The PF-L exhibited constant overheads for an all-read workload.*

Fig. 7a shows that the PF-L exhibited constant lock and unlock overheads of about $0.1\mu s$ across both sockets, while the PF-T overheads were on average $0.4\mu s$ on one socket, and up to $1.3\mu s$ on two sockets. This is attributable to the fact that in the PF-L, read lock and unlock operations only modify a single core-local variable. The PF-T read lock atomically increments a shared variable, which in turn invalidates other caches and bounces the variable across cores and sockets, yielding increased overhead.

As write percentages increase, read operations become more costly as *read_status* variables are read by writers on other cores and the write-related variables are constantly updated on all cores with read requests. This behavior also causes an increase in write overheads. The PF-T experienced higher overheads for read-dominant (Fig. 7b) and evenly distributed (Fig. 7c) workloads. Since all PF-T variables are on a single cache line, each update invalidates cache-line values for all other cores, resulting in an RMR for every entry and exit section.

▶ **Observation 5.** *For all workloads with some writes, overheads increased by up to 3× on two sockets.*

All insets in Fig. 7 show higher overheads on two sockets other than the PF-L for an all-read workload. This is attributable to higher cross-socket RMR latencies for both the PF-T and the PF-L for mixed workloads. Fig. 7a highlights the case in which reads in the PF-L generate no RMRs (by design) and does not exhibit increased overheads when executing on two sockets. This claim is further supported by throughput results in Fig. 6, where execution on two sockets consistently yielded lower throughput.

▶ **Observation 6.** *Reading under the PF-L incurred less overhead than reading under the PF-T.*

For all tested scenarios across varying write percentages and core counts, read operations under the PF-L yielded lower overheads than the PF-T. Fig. 7d shows trends in read overheads with varying write percentages. Beyond 50% writes, overheads were consistent for all read operations and at most $0.7\mu s$. The PF-L overheads for write-dominant workloads did not appreciably increase beyond 50% writes. With more writes, cache-line invalidations become frequent and cause higher overheads.

5 Schedulability Analysis of Phase-Fair Reader-Writer Locks

Recent work [9] presented an analysis framework for P-EDF built around a prior schedulability test [5]. Within this framework, each processor is analyzed in turn, incorporating the delays caused to the execution of tasks on that processor due to waiting for access to shared resources. In the discussion below, the processor under consideration is denoted P^*. The schedulability framework uses a fixed point iteration to bound the length of the analysis interval on P^*, which we denote $\mathcal{I}$, by using the concept of an *arrival curve (AC)* [26] and *processor demand criterion (PDC)* [5]. At each iteration, a bound on the delays over $\mathcal{I}$ caused by shared resources is required; this bound is what we must provide. Along with the original presentation of the analysis framework, an integer linear programming approach to bounding delays for mutex locks was given [9]. In order to apply this analysis framework to a system in which shared resources are instead managed by phase-fair reader/writer locks, we must instead provide bounds for that locking protocol. The schedulability framework is described in full detail in prior work [9], and the remainder of this section is devoted to determining a bound on delays under phase-fair reader/writer locks.

We build on the previously presented inflation-free analysis for mutex locks [9] to obtain such analysis. We begin by describing the types of delay, along with the constants and variables used in the formulation of our optimization problem. The remainder of the section is devoted to showing that the constraints we apply hold.

Types of delay. To check schedulability, analysis is required to bound *synchronization delay*, which includes delays due to both spinning and non-preemptive execution. *Spin delay* is the delay incurred on P^* when a task on P^* waits for a resource by spinning. *Arrival delay* is the delay on P^* that is incurred when a job is unable to begin executing due to the non-preemptive execution of a lower-priority job. Note that the job executing non-preemptively may be either spinning or executing with a satisfied request. Both types of blocking are illustrated in Fig. 2.

To constrain the computed arrival blocking, the inflation-free approach [9] leverages two key observations that are derived from existing schedulability analysis [5], generalized here:

- O1: Arrival blocking in an analysis interval $\mathcal{I}$ of length t is caused only by tasks with a relative deadline larger than t.
- O2: Only a single blocking request can cause arrival blocking.

In the rest of this section, we describe the creation of the optimization problem that we define for $\mathcal{I}$ to compute the maximum synchronization delay (denoted $B(P^*, t)$). This problem can be solved with a linear-programming solver, such as GLPK [1]. While we build on an existing framework, the assumptions that informed the construction of the approach for mutex locks do not all hold for phase-fair locks, which require new reasoning.

We begin by describing the constants and variables of our optimization problem. Then, we briefly describe the set of constraints that are straightforward modifications of the original approach; the proofs of these constraints are given in App. A. Finally, we present the constraints that require new reasoning unique to PF locks.

Constants and variables. We conduct schedulability analysis for each partition separately. Here, we focus on the analysis for partition P^*. We denote the set of all partitions by $\mathcal{P}$, and the set of all tasks by Γ. We refer to the set of tasks partitioned to a remote processor (any processor other than P^*) as Γ^r, and the set of tasks on a given processor P_k as $\Gamma(P_k)$. We denote the period of an arbitrary task, τ_i, by T_i, and its relative deadline by D_i. We reason about an arbitrary resource ℓ_q in the set of all resources Q. We use constants for the number of requests each job issues and the duration of requests by type; we denote the maximum duration of a read (resp., write) request issued by a job of τ_i for a resource ℓ_q with $L^R_{i,q}$ (resp., $L^W_{i,q}$). A job of τ_i issues at most $N^R_{i,q}$ (resp., $N^W_{i,q}$) read (resp., write) requests.

The following variables are used in our optimization problem to bound blocking:

- $X^{S,R}_{i,q}$ is the spin delay caused by read requests issued by τ_i for ℓ_q.
- $X^{S,W}_{i,q}$ is the spin delay caused by write requests issued by τ_i for ℓ_q.
- $X^{A,R}_{i,q}$ is the arrival blocking caused by read requests issued by τ_i for ℓ_q.
- $X^{A,W}_{i,q}$ is the arrival blocking caused by write requests issued by τ_i for ℓ_q.
- A^R_q is an indicator (*i.e.*, binary) variable. $A^R_q = 1$ indicates that arrival blocking is caused by a read request for ℓ_q, whereas $A^R_q = 0$ indicates that no arrival blocking is caused by a read request for ℓ_q.
- A^W_q is similarly an indicator of arrival blocking caused by a write request for ℓ_q.

For the specification of the optimization problem given below, we are applying the PDC. As such, the number of jobs τ_j on P^* (a local task) that must be considered during the analysis interval $\mathcal{I}$ of length t is $\mathit{nljobs}(\tau_j, t) = \left\lfloor \frac{t+T_j-D_j}{T_j} \right\rfloor$. The number of jobs of a remote task τ_j that must be accounted for is $\mathit{nrjobs}(\tau_j, t) = \left\lceil \frac{t+D_j}{T_j} \right\rceil$. The modifications described previously [9] allow for simple changes to the specification of the optimization problem to instead reason about the AC.

Optimization problem. The optimization problem we seek to solve is formulated to maximize the computed blocking subject to a set of constraints that limit this blocking by considering scenarios that cannot occur. This problem is as follows.

maximize $B(t) = \sum_{\forall \tau_i \in \Gamma} \sum_{\ell_q \in Q} [(X^{S,R}_{i,q} + X^{A,R}_{i,q}) \cdot L^R_{i,q} + (X^{S,W}_{i,q} + X^{A,W}_{i,q}) \cdot L^W_{i,q}]$

subject to the constrains in Tbl. 1.

Foundational RW constraints. The first set of constraints builds directly on the inflation-free analysis presented for mutex locks [9], with the distinction that we instead specify read- and write-versions of each variable, as detailed above. We describe these constraints briefly here and present the full versions in App. A.

Constraint (1) limits the computed arrival blocking terms for read and write requests by comparing the relative deadline of each task to the length of the deadline busy-period. Constraint (2) enforces that spin delay can be caused only by tasks remote to P^*. Constraints (3) and (4) limit the contribution of each request (read and write requests, resp.) to delays; each request can contribute to either arrival blocking or spin delay, but not both.

The next five constraints focus on arrival blocking. As arrival blocking can be caused by only a single request (Observation O2), it can be caused by either a read request or a write request (not both); this is enforced by Constraint (5). Constraints (6) and (7) leverage the fact that resources for which there are no read (resp., write) requests cannot cause read (resp., write) arrival blocking. Finally, Constraints (8) and (9) bound the total number of read (resp., write) requests that can cause arrival blocking by the binary variable indicating if arrival blocking is caused by a read (resp. write) request for that resource.

Helper variables. We introduce four helper variables, $X_{i,q}^{S,\text{R-to-W}}$, $X_{i,q}^{S,\text{R-to-R}}$, $X_{i,q}^{S,\text{W-to-W}}$, and $X_{i,q}^{S,\text{W-to-R}}$, to analyze the spin blocking caused by remote requests by cases. For example, $X_{i,q}^{S,\text{R-to-W}}$ is the number of read requests issued by τ_i that delay write requests.

Constraints on spin blocking. The following constraints limit the spin blocking that can be computed based on the possible interactions between read and write requests. These must account for the access patterns that can occur under phase-fair locks. Constraints (10) and (11) join the helper variables to those counting total read and write spin delay.

Proof of (10). Each read by a remote task τ_i can induce spin delay on a read request or a write request, but not both, on P^*, as all requests execute non-preemptively. Thus, the number of read requests of τ_i for ℓ_q that cause spin delay ($X_{i,q}^{S,R}$) is obtained by summing the number that delay read requests ($X_{i,q}^{S,\text{R-to-R}}$) and write requests ($X_{i,q}^{S,\text{R-to-W}}$), respectively. ◀

Proof of (11). Similar to that of Constraint (10). ◀

Constraints (12) and (13) limit the contribution of write requests to spin delay by considering the total number of read and write requests on P^* during $\mathcal{I}$.

Proof of (12). By Rules PF1 and PF3, a given read request may be delayed by at most one write phase. There are at most $\sum_{\tau_i \in \Gamma(P^*)} \mathit{nljobs}(\tau_i, t) \cdot N_{i,q}^{R}$ read requests for ℓ_q on P^* during $\mathcal{I}$. Thus, that number upper bounds the number of write requests from other processors that can cause delay to read requests for ℓ_q on P^*, which is $\sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{S,\text{W-to-R}}$. ◀

Proof of (13). There are at most $\sum_{\tau_i \in \Gamma(P^*)} \mathit{nljobs}(\tau_i, t) \cdot N_{i,q}^{W}$ write requests on P^* during $\mathcal{I}$. At most one write request per processor can delay the execution of each write request on P^*, because write requests are satisfied in FIFO order (by Rule PF2), requests execute non-preemptively, and only one write request is satisfied during each write phase (by Rule PF3). Thus, for each processor P_k, the number of write requests delaying write requests on P^* ($\sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{S,\text{W-to-W}}$) is bounded by the number of write requests on P^* in $\mathcal{I}$. ◀

Constraints (14) and (15) bound the impact of remote read requests on read requests on P^*. We use the following lemma and corollary in verifying them.

▶ **Lemma 7.** *A read request $\mathcal{R}_i^r$ is blocked by at most one read phase and one write phase.*

Proof. If there are no active requests when $\mathcal{R}_i^r$ is issued, it will be satisfied immediately.

If instead there are active read requests and no active write requests, $\mathcal{R}_i^r$ is satisfied immediately upon issuance by Rule PF4.

If there are active write requests and no active read requests when $\mathcal{R}_i^r$ is issued, it will be delayed by the current write phase and then satisfied after the currently satisfied write request completes by Rules PF1 and PF3.

If there is at least one active write request and one active read request when $\mathcal{R}_i^r$ is issued, then either a write request or a read request is currently satisfied. If a write request is satisfied, then $\mathcal{R}_i^r$ will be satisfied upon the completion of that request by Rules PF1 and PF3. If instead the resource is in a read phase, $\mathcal{R}_i^r$ must wait for the completion of this read phase (by Rule PF4) and the completion of a single write phase (by Rules PF1 and PF3).

Thus, in all cases, $\mathcal{R}_i^r$ is blocked by at most one read phase and one write phase. ◀

Table 1 Linear-program constraints. Constraints (1)–(9) are described in App. A.

Number	Constraint Specification
(1)	$\forall \tau \in \Gamma(P^*) \mid D_i \leq t,\ \forall \ell_q \in Q,\ X^{A,R}_{i,q} + X^{A,W}_{i,q} = 0$
(2)	$\sum_{\tau_i \in \Gamma(P^*)} \sum_{\ell_q \in Q} X^{S,R}_{i,q} + X^{S,W}_{i,q} = 0$
(3)	$\forall \tau_i \in \Gamma^r,\ \forall \ell_q \in Q,\ X^{S,R}_{i,q} + X^{A,R}_{i,q} \leq \mathit{nrjobs}(\tau_i, t) \cdot N^R_{i,q}$
(4)	$\forall \tau_i \in \Gamma^r,\ \forall \ell_q \in Q,\ X^{S,W}_{i,q} + X^{A,W}_{i,q} \leq \mathit{nrjobs}(\tau_i, t) \cdot N^W_{i,q}$
(5)	$\sum_{\ell_q \in Q} A^R_q + A^W_q \leq 1$
(6)	$\forall \ell_q \in Q,\ A^R_q \leq \sum_{\tau_i \in \Gamma(P^*) \mid D_i > t} N^R_{i,q}$
(7)	$\forall \ell_q \in Q,\ A^W_q \leq \sum_{\tau_i \in \Gamma(P^*) \mid D_i > t} N^W_{i,q}$
(8)	$\forall \ell_q \in Q,\ \sum_{\forall \tau_i \in \Gamma(P^*)} X^{A,R}_{i,q} \leq A^R_q$
(9)	$\forall \ell_q \in Q,\ \sum_{\forall \tau_i \in \Gamma(P^*)} X^{A,W}_{i,q} \leq A^W_q$ Constraints adapted from [9]
(10)	$\forall \tau_i \in \Gamma^r,\ \forall \ell_q \in Q,\ X^{S,R}_{i,q} = X^{S,\text{R-to-R}}_{i,q} + X^{S,\text{R-to-W}}_{i,q}$ New Constraints
(11)	$\forall \tau_i \in \Gamma^r,\ \forall \ell_q \in Q,\ X^{S,W}_{i,q} = X^{S,\text{W-to-R}}_{i,q} + X^{S,\text{W-to-W}}_{i,q}$
(12)	$\forall \ell_q \in Q,\ \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X^{S,\text{W-to-R}}_{x,q} \leq \sum_{\tau_i \in \Gamma(P^*)} \mathit{nljobs}(\tau_i, t) \cdot N^R_{i,q}$
(13)	$\forall P_k \in \mathcal{P},\ \forall \ell_q \in Q,\ \sum_{\tau_x \in \Gamma(P_k)} X^{S,\text{W-to-W}}_{x,q} \leq \sum_{\tau_i \in \Gamma(P^*)} \mathit{nljobs}(\tau_i, t) \cdot N^W_{i,q}$
(14)	$\forall \ell_q \in Q,\ \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X^{S,\text{R-to-R}}_{x,q} \leq \sum_{\tau_i \in \Gamma(P^*)} \mathit{nljobs}(\tau_i, t) \cdot N^R_{i,q}$
(15)	$\forall \ell_q \in Q,\ \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X^{S,\text{R-to-R}}_{x,q} \leq \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X^{S,\text{W-to-R}}_{i,q}$
(16)	$\forall \ell_q \in Q,\ \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X^{S,\text{R-to-W}}_{x,q}$ $\leq \sum_{\tau_i \in \Gamma(P^*)} (\mathit{nljobs}(\tau_i, t) \cdot N^W_{i,q}) + \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X^{S,\text{W-to-W}}_{x,q}$
(17)	$\forall \tau_x \in \Gamma^R,\ \forall \ell_q \in Q,\ X^{S,\text{W-to-W}}_{x,q} \leq \mathit{nrjobs}(\tau_x, t) \cdot \sum_{\tau_i \in \Gamma(P^*)} (\mathit{nrjobs}(\tau_i, D_x) \cdot N^W_{i,q})$
(18)	$\forall \tau_x \in \Gamma^R,\ \forall \ell_q \in Q,\ X^{S,\text{W-to-R}}_{x,q} \leq \mathit{nrjobs}(\tau_x, t) \cdot \sum_{\tau_i \in \Gamma(P^*)} (\mathit{nrjobs}(\tau_i, D_x) \cdot N^R_{i,q})$
(19)	$\forall \tau_x \in \Gamma^R,\ \forall \ell_q \in Q,\ X^{S,\text{R-to-R}}_{x,q} \leq \mathit{nrjobs}(\tau_x, t) \cdot \sum_{\tau_i \in \Gamma(P^*)} (\mathit{nrjobs}(\tau_i, D_x) \cdot N^R_{i,q})$
(20)	$\forall \ell_q \in Q,\ A^R_q + A^W_q = 0 \Rightarrow \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X^{A,R}_{x,q} + X^{A,W}_{x,q} \leq 0$
(21)	$\forall \ell_q \in Q,\ A^R_q = 1 \Rightarrow \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X^{A,R}_{x,q} \leq 1$
(22)	$\forall \ell_q \in Q,\ A^R_q = 1 \Rightarrow \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X^{A,W}_{x,q} \leq 1$
(23)	$\forall \ell_q \in Q,\ A^R_q = 1 \Rightarrow \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X^{A,R}_{x,q}$ $\leq \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X^{A,W}_{x,q}$
(24)	$\forall \ell_q \in Q,\ A^W_q = 1 \Rightarrow \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X^{A,R}_{x,q}$ $\leq 1 + \sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X^{A,W}_{x,q}$
(25)	$\forall P_k \neq P^*,\ \forall \ell_q \in Q,\ A^W_q = 1 \Rightarrow \sum_{\tau_x \in \Gamma(P_k)} X^{A,W}_{x,q} \leq 1$

▶ **Corollary 8.** *If a read request $\mathcal{R}^r_i$ is blocked by W write requests, it is blocked by at most W read phases.*

Proof. In the proof of Lemma 7, which enumerated all possible blocking scenarios for a read request $\mathcal{R}^r_i$, the only scenario in which a request $\mathcal{R}^r_i$ is blocked by a read request is when a write request also blocks $\mathcal{R}^r_i$. ◀

Proof of (14). During $\mathcal{I}$, there are at most $\sum_{\tau_i \in \Gamma(P^*)} \mathit{nljobs}(\tau_i, t) \cdot N^R_{i,q}$ read requests on P^*. By Lemma 7, each read requests can be delayed by at most one read phase. Thus the total number of read requests that cause spin blocking for read requests on P^* ($\sum_{P_k \in \mathcal{P} \mid P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X^{S,\text{R-to-R}}_{x,q}$) is bounded by the number of read requests on P^*. ◀

Proof of (15). By Cor. 8, a read request can be delayed by a read phase only if it is also delayed by a write phase. Thus, the total number of read requests causing spin blocking for read requests on P^* ($\sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{S,\text{R-to-R}}$) is bounded by the total number of write requests causing spin blocking for those requests ($\sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{i,q}^{S,\text{W-to-R}}$). ◀

We now examine how read requests can delay write requests.

▶ **Lemma 9.** *If W write phases block a write request $\mathcal{R}_i^w$, at most $W+1$ read phases block $\mathcal{R}_i^w$.*

Proof. By Rule PF1, read phases and write phases alternate. Before each of the W write phases that block $\mathcal{R}_i^w$, a read phase can occur. Additionally, after the last blocking write phase and before the satisfaction of $\mathcal{R}_i^w$, an additional read phase can occur. Therefore, at most $W+1$ read phases can block $\mathcal{R}_i^w$. ◀

Constraint (16) limits read-to-write blocking and its proof leverages Lemma 9.

Proof of (16). There are $\sum_{\tau_i \in \Gamma(P^*)} (\mathit{nljobs}(\tau_i, t) \cdot N_{i,q}^W)$ write request to consider on P^* during $\mathcal{I}$. For each of these requests individually, if some number W write phases block the request, up to $W+1$ read phases can also block that request, by Lemma 9. In total, $\sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{S,\text{W-to-W}}$ write requests can block these write requests, by definition. As each write request can incur one additional blocking by a read request, an additional $\sum_{\tau_i \in \Gamma(P^*)} (\mathit{nljobs}(\tau_i, t) \cdot N_{i,q}^W)$ read requests can block write requests on P^*. Thus, in total the number of read requests that can delay write requests ($\sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{S,\text{R-to-W}}$) is bounded by $\sum_{\tau_i \in \Gamma(P^*)} (\mathit{nljobs}(\tau_i, t) \cdot N_{i,q}^W) + \sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{S,\text{W-to-W}}$. ◀

Finally, we constrain the impact of each remote task on tasks on P^* by considering how jobs may overlap based on their respective periods and deadlines. The next two lemmas are used in verifying Constraints (17)–(19).

▶ **Lemma 10.** *The requests for ℓ_q issued by a single job of a remote task $\tau_x \in \Gamma^r$ overlap with at most $\mathit{nrjobs}(\tau_i, D_x) \cdot N_{i,q}^w$ write requests for ℓ_q issued by jobs of $\tau_i \in \Gamma(P^*)$.*

Proof. The number of jobs of τ_i that overlap with a single job of τ_x is at most $\mathit{nrjobs}(\tau_i, D_x)$. Each job of τ_i issues up to $N_{i,q}^w$ write requests. Thus, the requests from a single job of $\tau_x \in \Gamma^r$ overlap with at most $\mathit{nrjobs}(\tau_i, D_x) \cdot N_{i,q}^w$ write requests for ℓ_q issued by jobs of τ_i. ◀

▶ **Lemma 11.** *The requests for ℓ_q issued by a single job of a remote task $\tau_x \in \Gamma^r$ overlap with at most $\mathit{nrjobs}(\tau_i, D_x) \cdot N_{i,q}^r$ read requests for ℓ_q issued by jobs of $\tau_i \in \Gamma(P^*)$.*

Proof. Follows as above, but for read requests. ◀

Constraint (17) limits blocking caused by write requests.

Proof of (17). By Lemma 10, a single job of a task $\tau_x \in \Gamma^r$ overlaps with up to $\mathit{nrjobs}(\tau_i, D_x) \cdot N_{i,q}^w$ write requests of an arbitrary task $\tau_i \in \Gamma(P^*)$. Thus, a single job of τ_x can overlap with a total of $\sum_{\tau_i \in \Gamma(P^*)} (\mathit{nrjobs}(\tau_i, D_x) \cdot N_{i,q}^W)$ write requests issued on P^*. Because of the non-preemptive execution and FIFO satisfaction order of write requests (Rule PF2), each of these write requests on P^* can be delayed by at most one overlapping write request per job of a remote task. During $\mathcal{I}$, $\mathit{nrjobs}(\tau_x, t)$ jobs of τ_x must be considered. Thus, the total number of write requests of τ_x that can cause spin delay on P^* ($X_{x,q}^{S,\text{W-to-W}}$) is bounded by $\mathit{nrjobs}(\tau_x, t) \cdot \sum_{\tau_i \in \Gamma(P^*)} (\mathit{nrjobs}(\tau_i, D_x) \cdot N_{i,q}^W)$. ◀

Constraints (18) and (19) limit blocking caused to read requests on P^*.

Proof of (18). Lemma 11 bounds the number of read requests that a job of $\tau_x \in \Gamma^r$ may overlap with. By Lemma 7, at most one write phase can delay each read request, implying that at most one write request per job can delay each read request. Thus, the constraint follows similarly to Constraint (17). ◀

Proof of (19). Follows similarly to Constraint (18) by instead applying that each read request can be blocked by at most one read request (by Lemma 7). ◀

Constraints on arrival blocking. A single request on P^* can cause arrival blocking by its non-preemptive blocking and then execution. The duration of this arrival blocking is impacted by the type of request that causes it.

The following constraints are indicator constraints; if a variable in the optimization problem holds a specified value, an additional constraint is imposed. Some linear programming solvers allow the direct specification of indicator constraints. Alternatively, each indicator constraint can be converted to a set of linear constraints by using Big-M techniques [10].

Constraint (20) accounts for the case in which no requests for ℓ_q cause arrival blocking.

Proof of (20). If neither a read request nor a write request for ℓ_q can cause arrival blocking ($A_q^R + A_q^W = 0$), the total number of remote requests that can contribute to arrival blocking ($\sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{A,R} + X_{x,q}^{A,W}$) is 0. ◀

Because any arrival blocking is caused by a single request on P^*, we apply reasoning based on request type to eliminate blocking that cannot possibly occur. Constraints (21)–(23) apply if a read request causes arrival blocking. Recall that a single read request can be blocked by at most one read request and one write request by Lemma 7.

Proof of (21). If a read request on P^* causes arrival blocking ($A_q^R = 1$), at most one read phase can contribute to its delay by Lemma 7. Thus, the total number of read requests from remote processors that can cause arrival blocking ($\sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{A,R}$) is bounded by 1. ◀

Proof of (22). Similarly, the total number of write requests from remote processors that can cause arrival blocking ($\sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{A,W}$) is bounded by 1. ◀

Proof of (23). If a read request on P^* causes arrival blocking ($A_q^R = 1$), then by Cor. 8, if it is blocked by W write requests, it will be blocked by at most W read requests. Because of the non-preemptive execution of requests, any requests that contribute to the blocking of the read on P^* that causes arrival blocking are requests issued by tasks on remote processors. Thus, the total number of write requests that block this read request ($\sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{A,W}$) upper bounds the number of read requests that block this read request ($\sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{A,R}$). ◀

Constraints (24) and (25) consider arrival blocking that is caused by a write request.

Proof of (24). If a write request on P^* causes arrival blocking ($A_q^W = 1$), the number of read requests that can block it is bounded by one more than the write requests causing delay (by Lemma 9). Thus, the total number of remote read requests that cause arrival blocking ($\sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{A,R}$) is bounded by one more than the number write requests on remote processors that cause arrival blocking ($\sum_{P_k \in \mathcal{P} | P_k \neq P^*} \sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{A,W}$). ◀

Table 2 Summary of percentage TSA improvement of LP:PF.

	Min	Q1	Median	Q3	Max
Inflation:PF	0.383	10.1	28.5	54.5	158.6

Proof of (25). If a write request on P^* causes arrival blocking ($A_q^W = 1$), by Rules PF2 and PF3 and the non-preemptive execution of requests, at most one write request per remote processor can delay that write request, as requests execute non-preemptively. Thus, the total number of write requests that cause arrival blocking issued by tasks on P_k ($\sum_{\tau_x \in \Gamma(P_k)} X_{x,q}^{A,W}$) is bounded by 1. ◀

6 Schedulability Evaluation

To explore the benefit of our new approaches we conducted a schedulability study by using the SchedCAT toolkit [3] and building upon a prior implementation [9].

Schedulability improvements. We begin by comparing our inflation-free analysis for phase-fair reader-writer locking protocols (labeled "LP:PF") to the existing per-request inflation-based PF analysis (labeled "Inflation:PF"). To reduce the time it took to compute schedulability, we applied our holistic analysis for phase-fair locks only if the per-request inflation-based approach failed to be schedulable. The line labeled "NOLOCK" shows the computed schedulability if the delays for resource accesses are ignored.

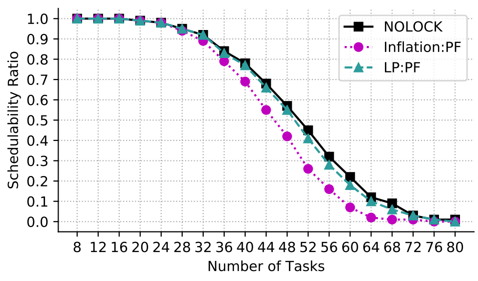
In this study, we computed schedulability for increasing task counts under different scenarios, with 216 scenarios total. Each scenario is a different combination of certain system parameters. We considered a system with eight processors. Task periods were selected from a log-uniform distribution in $[10ms, 100ms]$ or in $[1ms, 1000ms]$. Each task's utilization was chosen from an exponential distribution with a mean of 0.1. The number of resources (n_r) in a scenario was selected from $\{4, 8, 16\}$. For each resource, the probability that a task requires that resource was chosen from $\{0.1, 0.25, 0.5\}$. The number of times a task accesses a given resource was either 1 or was selected from $\{1, \ldots, 5\}$. For a given access to be write access (instead of a read access) was chosen with a probability selected from $\{0.01, 0.1, 0.5\}$. Request durations were either short (selected uniformly from $[1\mu s, 25\mu s]$) or medium (selected uniformly from $[25\mu s, 100\mu s]$). These parameters closely reflect those on which the original holistic analysis framework [9] was analyzed.

This study resulted in 216 schedulability graphs (one per scenario), which show the ratio of schedulable tasks systems out of the 1,000 systems generated for each data point. Performance is evaluated on the basis of *task schedulable area (TSA)*, the area under a given curve as computed by a midpoint Riemann sum. In Tbl. 2, we summarize the data on the percentage TSA improvement of LP:PF, and we highlight some key scenarios in Fig. 8.

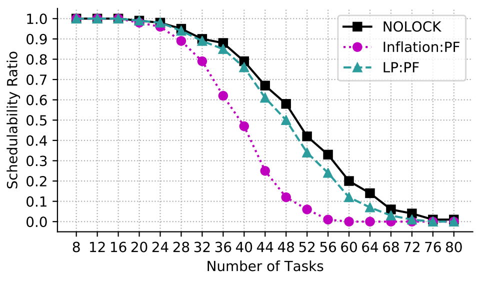
▶ **Observation 12.** *The LP:PF approach always resulted in a higher TSA than Inflation:PF.*

This is illustrated in Fig. 8. The cases in which LP:PF resulted in the largest percentage improvement (50.% to 158.6%) were primarily for scenarios with write probability of 0.1 or 0.5; 96.9% of these scenarios had write probability of 0.1 or 0.5.

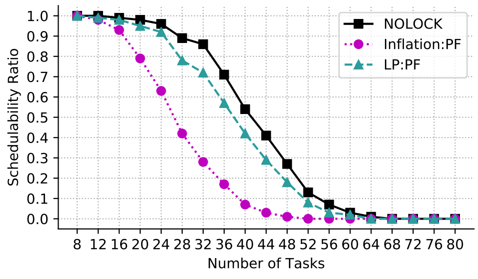
▶ **Observation 13.** *In some scenarios, the LP:PF resulted in only small increases in schedulability.*



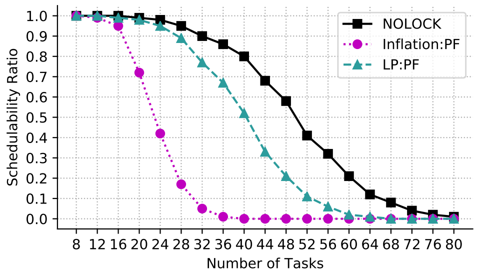
(a) Near Q1 (10.1% improvement)



(b) Near median (28.5% improvement)



(c) Near Q3 (54.5% improvement)



(d) Significant improvements (110%)

Figure 8 Comparisons against Inflation:PF.

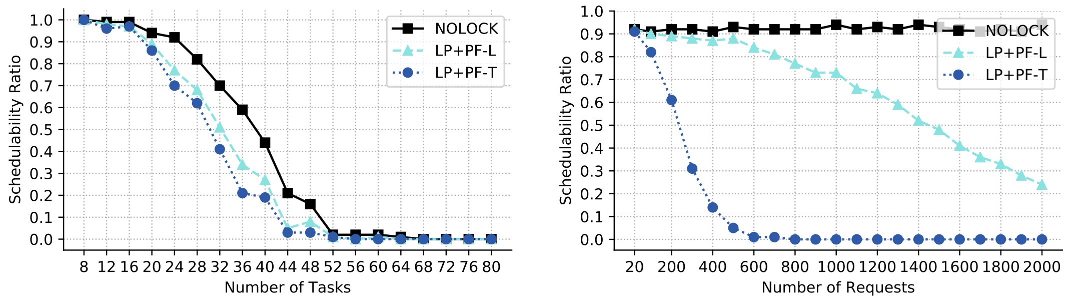
This is illustrated in Tbl. 2 and Fig. 8a. For the scenarios with TSA improvements in the first quartile, in which the LP:PF had a small percentage of improvement, both approaches tended to yield a TSA close to that of NOLOCK.

Overhead-aware schedulability. We conducted an additional scheduability study in which we incorporated protocol overheads. We inflated requests by the corresponding overhead and analyzed the resulting systems with our PF analysis; "LP+PF-T" (resp., "LP+PF-L") represents the computed schedulability with the PF-T (resp., PF-L) overheads added. We measured overheads as described in Sec. 4 with eight cores across two sockets for scenarios with up to 10% write requests. For the PF-T, this resulted in read (resp., write) overhead of 2.2μs (resp., 1.7μs), and for the PF-L, read (resp., write) overhead of 0.5μs (resp., 0.9μs).

In our study of read-dominated workloads (write probability in {0.01, 0.1}), we observed moderate differences, with an average TSA improvement for LP+PF-L of 1.01%. In some scenarios, the overhead was negligible relative to the blocking. In others, generally those with more resource accesses, the TSA difference was more pronounced. We observed scenarios with up to a 10.4% improvement, as depicted in Fig. 9a. These results support the following.

▶ **Observation 14.** *For read-dominant workloads, our new PF-L protocol and schedulability analysis dominated prior state-of-the art approaches.*

The schedulability improvements initially seemed modest relative to the impacts of lower overhead on throughput. However, the task systems considered in Sec. 4 are quite different (*e.g.*, significant execution time spent in the execution of requests) from those detailed in the schedulability study just discussed. Therefore, to assess the impact of overheads alone (without blocking) in a system with significant resource requirements, we conducted an additional overhead-aware schedulability study that focused on read-only workloads with a variable number of requests for a single shared resource. Here, we applied overheads measured from a system with 0% write requests; thus, we applied overheads of 1.2μs for



(a) Periods selected from {1,1000}ms, 16 resources, 0.5 access probability, up to 5 requests per resource, short request durations, and write probability of 0.01.

(b) 32 tasks with periods selected from {1,1000}ms, one resource, access probability of 1.0, and write probability of 0.

Figure 9 Schedulability with protocol overhead incorporated.

the PF-T and 0.2μs for the PF-L. Fig. 9b gives the schedulability graph that resulted from this study. These findings are consistent with the throughput experiments (*e.g.*, Fig. 1), and confirm that small overheads can significantly affect throughput and schedulability for synchronization-heavy read-dominant workloads.

7 Conclusion

We presented a new phase-fair reader/writer lock implementation and inflation-free PF schedulability analysis that, taken together, can improve both throughput and schedulability in comparison to prior alternatives when supporting read-mostly workloads. While this work was motivated by heavily read-dominant workloads, our findings suggest that the presented lock implementation may be competitive, if not superior, to previous RW locking protocols in most applications. We have demonstrated these improvements via experiments on real hardware and via a schedulability study. In future work, we intend to explore how other concurrent algorithms can be adapted based on cache coherence and performance properties to improve scalability similar to that we have demonstrated herein.

References

1 GLPK (GNU Linear Programming Kit). `https://www.gnu.org/software/glpk/`.

2 LITMUS^{RT} home page. `http://www.litmus-rt.org/`.

3 SchedCAT: Schedulability test collection and toolkit. `https://github.com/brandenburg/schedcat`, 2020. Accessed: 2020-06-21.

4 S. Altmeyer, R. Douma, W. Lunniss, and R. Davis. Evaluation of cache partitioning for hard real-time systems. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems*, 2014.

5 S. Baruah. Resource sharing in EDF-scheduled systems: A closer look. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, 2006.

6 A. Bastoni, B. Brandenburg, and J. Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. *Proceedings of the 6th Workshop on Operating Systems Platforms for Embedded Real-Time applications*, 2010.

7 V. Bhatt and P. Jayanti. Constant RMR solutions to reader writer synchronization. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 2010.

8 V. Bhatt and P. Jayanti. Specification and constant RMR algorithm for phase-fair reader-writer lock. In *Proceedings of the 12th International Conference on Distributed Computing and Networking*, 2011.

9 A. Biondi and B. Brandenburg. Lightweight real-time synchronization under P-EDF on symmetric and asymmetric multiprocessors. In *Proceedings of the 28th Euromicro Conference on Real-Time Systems*, 2016.

10 S. Bradley, A. Hax, and T. Magnanti. Applied mathematical programming, Chapter 9 (Addison-Wesley, 1977). `http://web.mit.edu/15.053/www/AMP-Chapter-09.pdf`, 2021.

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

12 B. Brandenburg and J. Anderson. Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-Time Systems*, 46(1):25–87, 2010.

13 B. Brandenburg and J. Anderson. Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In *Proceedings of the 9th ACM International Conference on Embedded Software*, 2011.

14 M. Campoy, A.P. Ivars, and J.V. Busquets-Mataix. Static use of locking caches in multitask preemptive real-time systems. In *IEEE/IEE Real-Time Embedded Systems Workshop*, 2001.

15 M. Chisholm, B. Ward, N. Kim, and J. Anderson. Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *Proceedings of the 36th IEEE International Real-Time Systems Symposium*, December 2015.

16 P. Courtois, F. Heymans, and D. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667–668, 1971.

17 James R Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, 1983.

18 J. Hennessy and D. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

19 M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

20 J. Herter, P. Backes, F. Haupenthal, and J. Reineke. CAMA: A predictable cache-aware memory allocator. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, 2011.

21 H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, 2013.

22 D. Kirk and J. Strosnider. SMART (strategic memory allocation for real-time) cache design using the MIPS R3000. In *Proceedings of the 11th Real-Time Systems Symposium*, 1990.

23 Y. Lev, V. Luchangco, and M. Olszewski. Scalable reader-writer locks. In *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures*, 2009.

24 P. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, Beaverton, OR, 2004.

25 J. Mellor-Crummey and M. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the 3rd ACM Symposium on Principles and Practice of Parallel Programming*, 1991.

26 L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, 2000.

27 B. Ward. Relaxing resource-sharing constraints for improved hardware management and schedulability. In *Proceedings of the 36th International IEEE Real-Time Systems Symposium*, 2015.

28 B. Ward and J. Anderson. Multi-resource real-time reader/writer locks for multiprocessors. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*, 2014.

29 B. Ward, J. Herman, C. Kenna, and J. Anderson. Making shared caches more predictable on multicore platforms. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, 2013.

30 A. Wieder and B. Brandenburg. On spin locks in AUTOSAR: Blocking analysis of FIFO, unordered, and priority-ordered spin locks. In *Proceedings of the 34th IEEE International Real-Time Systems Symposium*, 2013.

31 M. Xu, L. T. X. Phan, H.-Y. Choi, and I. Lee. Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium*, 2016.

32 M. Xu, L. T. X. Phan, H.-Y. Choi, and I. Lee. vCAT: Dynamic cache management using CAT virtualization. In *Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium*, 2017.

33 J.-H. Yang and J. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, August 1995.

34 H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2014.

A Additional Constraints

Constraints (1)-(9), proven here, are similar to constraints in prior inflation-free analysis [9].

Proof of (1). Follows directly from Observation O1. ◀

Proof of (2). In order to cause spin delay, a local task must have a satisfied request while another request is blocked. However, because tasks spin and execute critical sections non-preemptively, there can be at most one active request on P^* at any given time. Therefore, tasks on P^* cannot cause spin delay; only remote tasks can cause spin delay. ◀

The following lemma can be proven using reasoning on the behavior of the PF lock similar to that used to prove Lemmas 7 and 9.

▶ **Lemma 15.** *Each remote request $\mathcal{R}_x$ can contribute to delaying requests on P^* at most once, and that delay is realized as either arrival blocking or spin delay, but not both.*

Proof of (3) and (4). Both follow from Lemma 15. ◀

Constraints (5)–(9) concern arrival blocking.

Proof of (5). Follows from Observation O2; only one request can cause arrival blocking, and each request is only for a single resource and is either a read request or a write request. ◀

Proof of (6). Recall that A_q^R is a binary indicator variable. By Observation O1, arrival blocking is only caused by tasks with a relative deadline larger than t. If no read request for ℓ_q is issued by any task with a deadline greater than t (*i.e.*, the sum on the right-hand side is 0), then is is not possible to have a read request for ℓ_q cause arrival blocking. ◀

Proof of (7). Similarly, we constrain arrival blocking due to a write request. ◀

Proof of (8). For each resource ℓ_q, the number of read requests that can cause arrival blocking is upper-bounded by A_q^R; at most one request can cause arrival blocking (Observation O2), and if $A_q^R = 0$, no request for that resource can cause arrival blocking. ◀

Proof of (9). Similarly, the arrival blocking caused by write requests is constrained. ◀