

Spin-Based Reader-Writer Synchronization for Multiprocessor Real-Time Systems*

Björn B. Brandenburg

James H. Anderson

July 2010

Abstract

Reader preference, writer preference, and task-fair reader-writer locks are shown to cause undue blocking in multiprocessor real-time systems. Phase-fair reader writer locks, a new class of reader-writer locks, are proposed as an alternative. Three local-spin phase-fair lock algorithms, one with constant remote-memory-reference complexity, are presented and demonstrated to be efficiently implementable on common hardware platforms. Both task- and phase-fair locks are evaluated and contrasted to mutex locks in terms of hard and soft real-time schedulability—each under both global and partitioned scheduling—under consideration of runtime overheads on a multicore Sun “Niagara” UltraSPARC T1 processor. Formal bounds on worst-case blocking are derived for all considered lock types.

1 Introduction

With the transition to multicore architectures by most (if not all) major chip manufacturers, multiprocessors are now a standard deployment platform for (soft) real-time applications. This has led to renewed interest in real-time multiprocessor scheduling and synchronization algorithms (see (Brandenburg and Anderson, 2008; Brandenburg et al., 2008a,b; Calandrino et al., 2006) for recent comparative studies and relevant references). However, prior work on synchronization has been somewhat limited, being mostly focused on mechanisms that ensure strict *mutual exclusion* (mutex). *Reader-writer* (RW) synchronization, which requires mutual exclusion only for updates (and not for reads) has not been considered in prior work on real-time shared-memory multiprocessor systems despite its great practical relevance.

The need for RW synchronization arises naturally in many situations. Two common examples are few-producers/many-consumers relationships (*e.g.*, obtaining and distributing sensor data) and rarely-changing shared state (*e.g.*, configuration information). As an example for the former, consider a robot such as TU Berlin’s autonomous helicopter Marvin (Musial et al., 2006): its GPS receiver updates the current position estimate 20 times per second, and the latest position estimate is read at various rates by a flight controller and by image acquisition, camera

*Work supported by IBM, Intel, and Sun Corps.; NSF grants CNS 0834270, CNS 0834132, and CNS 0615197; and ARO grant W911NF-06-1-0425.

targeting, and communication modules. An example of rarely-changing shared data occurs in work of Gore et al. (Gore et al., 2004), who employed RW locks to optimize latency in a real-time notification service. In their system, every incoming event must be matched against a shared subscription lookup table to determine the set of subscribing clients. Since events occur very frequently and changes to the table occur only very rarely, the use of a regular mutex lock “can unnecessarily reduce [concurrency] in the critical path of event propagation.” (Gore et al., 2004)

In practice, RW locks are in wide-spread use since they are supported by all POSIX-compliant real-time operating systems. However, to the best of our knowledge, such locks have not been analyzed in the context of multiprocessor real-time systems from a schedulability perspective. In fact, RW locks that are subject to potential starvation have been suggested to practitioners without much concern for schedulability (Li and Yao, 2003). This highlights the need for a well-understood, analytically-sound real-time RW synchronization protocol.

Related work. In work on non-real-time systems, Courtois et al. were the first to investigate RW synchronization and proposed two semaphore-based RW locks (Courtois et al., 1971): a *writer preference lock*, wherein writers have higher priority than readers, and a *reader preference lock*, wherein readers have higher priority than writers. Both are problematic for real-time systems as they can give rise to extended delays and even starvation. To improve reader throughput at the expense of writer throughput in distributed systems, Andrews proposed a token passing protocol under which only a single token is required for reading but all tokens must be obtained prior to writing (Andrews, 1991). Using a conceptually similar approach, Hsieh and Weihl proposed a semaphore-based RW lock for large shared-memory systems wherein the lock state is distributed across processors (Hsieh and Weihl, 1992).

In work on scalable synchronization on shared-memory multiprocessors, Mellor-Crummey and Scott proposed spin-based reader preference, writer preference, and *task-fair* RW locks (Mellor-Crummey and Scott, 1991b). In a task-fair RW lock, readers and writers gain access in strict FIFO order, which avoids starvation. In the same work, Mellor-Crummey and Scott also proposed *local-spin* versions of their RW locks, in which excessive memory bus traffic under high contention is avoided. An alternative local-spin implementation of task-fair RW locks was later proposed by Krieger et al. (Krieger et al., 1993).

McKenney introduced throughput-oriented criteria for selecting locking primitives and characterized the tradeoffs of task-fair and distributed RW locks in this context (McKenney, 1996). A probabilistic performance analysis of task-fair RW locks wherein reader arrivals are modeled as a Poisson process was conducted by Reiman and Wright (Reiman and Wright, 1991).

In work on uniprocessor real-time systems, two relevant suspension-based protocols have been proposed: Baker’s stack resource policy (Baker, 1991) and Rajkumar’s read-write priority ceiling protocol (Rajkumar, 1991). The latter has also been studied in the context of distributed real-time databases (Rajkumar, 1991).

An alternative to locking is the use of specialized *non-blocking* algorithms (Anderson and Holman, 2000). However, compared to lock-based RW synchronization, which allows in-place updates, non-blocking approaches usually require additional memory, incur significant copying or retry overheads, and are less general.¹ In this paper, we focus on lock-based RW syn-

¹In non-blocking read-write algorithms, the value to be written must be pre-determined and cannot depend on the current state of the shared object. With locks, an update can be computed based on the current value.

chronization in cache-coherent shared-memory multiprocessor systems.

Contributions. We present the first systematic study of RW synchronization in the context of multiprocessor real-time systems. Specifically, we

- examine the applicability of the above-mentioned existing RW lock types to real-time systems (Section 3.2) and demonstrate that neither task-fair RW locks nor preference locks reliably reduce worst-case blocking (Sections 3.2.2 and 3.2.3);
- identify the characteristics that restrict the effectiveness of said locks and propose a novel “phase-fair” RW lock to avoid these limitations (Section 3.2.4);
- show that reader blocking bounds are less (asymptotically) with phase-fair RW locks than with task-fair RW locks (Section 3.3) and prove analytical bounds on the worst-case delay of sporadic tasks due to blocking under task-fair mutex locks (Section A.4), reader and writer preference locks (Section A.5), task-fair RW locks (Section A.7), and phase-fair RW locks (Section A.8);
- give three efficient implementations of phase-fair RW locks (Section 4) specialized for shared-cache, private-cache, and memory-constrained multiprocessors; and
- report on the results of an extensive performance evaluation of RW synchronization choices in terms of hard and soft real-time schedulability under consideration of system overheads (Section 5).

Our experiments show that, for the considered real-time workloads on our test platform, it is beneficial to employ RW locks in general, and that employing phase-fair RW locks can significantly improve real-time performance if (approximately) at most 25%–30% of all critical sections are writes.

We formalize our system model and summarize relevant background next.

2 System Model

We consider the classic problem (Liu and Layland, 1973) of scheduling a system of n *sporadic tasks*, T_1, \dots, T_n , on m identical processors. Each task T_i is specified by its *worst-case execution cost*, $e(T_i)$, its *period*, $p(T_i)$, and its *relative deadline*, $d(T_i)$, where $d(T_i) \geq e(T_i) > 0$. Task T_i ’s *utilization* (or *weight*) reflects the processor share that it requires and is given by $e(T_i)/p(T_i)$.

The j^{th} *job* (or *invocation*) of task T_i is denoted T_i^j . Such a job T_i^j becomes available for execution at its *release time* (or *arrival time*), $a(T_i^j)$. The spacing between job releases must satisfy $a(T_i^{j+1}) \geq a(T_i^j) + p(T_i)$. T_i is further said to be *periodic* if $a(T_i^{j+1}) = a(T_i^j) + p(T_i)$ holds for any j .

T_i^j is *pending* from time $a(T_i^j)$ until it *completes* at time $c(T_i^j)$. A job T_i^j should complete execution by its *absolute deadline*, $a(T_i^j) + d(T_i)$, and is *tardy* if it completes later. While pending, a job is either *preemptable* or *non-preemptable*. A job scheduled on a processor can only be preempted when it is preemptable.

2.1 Multiprocessor Scheduling

There are two fundamental approaches to scheduling sporadic tasks on multiprocessors — *global* and *partitioned*. With global scheduling, processors are scheduled by selecting jobs from a single, shared queue, whereas with partitioned scheduling, each processor has a private queue and is scheduled independently using a uniprocessor scheduling policy (hybrid approaches exist, too (Anderson et al., 2005; Baker and Baruah, 2007; Calandrino et al., 2007)). Tasks are statically assigned to processors under partitioning. As a consequence, under partitioned scheduling, all jobs of a task execute on the same processor, whereas *migrations* may occur in globally-scheduled systems. A discussion of the tradeoffs between global and partitioned scheduling is beyond the scope of this paper and the interested reader is referred to prior studies concerning such tradeoffs (Brandenburg et al., 2008a; Calandrino et al., 2006).

We consider one representative algorithm from each category: in the partitioned case, the *partitioned EDF* (P-EDF) algorithm, wherein the *earliest-deadline-first* (EDF) algorithm is used on each processor, and in the global case, the *global EDF* (G-EDF) algorithm. However, the synchronization algorithms considered herein do not depend on the scheduling algorithm in use, and the analysis presented in Appendix A applies equally to other scheduling algorithms that assign each job a fixed priority, including partitioned static-priority (P-SP) scheduling. Further, we consider both *hard* real-time (HRT) systems in which deadlines should not be missed, and *soft* real-time systems (SRT) in which bounded deadline tardiness is permissible.

We let $r(T_i)$ denote a bound on task T_i 's *worst-case response time* or *maximum relative completion time*, *i.e.*, a bound on the maximum time that any job of task T_i can be pending. In HRT systems, if a task system is schedulable, then by definition $r(T_i) = d(T_i)$.² In SRT systems, $r(T_i)$ is given by the sum of $d(T_i)$ and T_i 's tardiness bound (if it exists) (Devi and Anderson, 2008; Leontyev and Anderson, 2007).

2.2 Resources

When a job T_i^j intends to update (observe) the state of a *resource* \mathcal{L} , it *issues* a *write* (*read*) *request* $\mathcal{W}_{\mathcal{L}}$ ($\mathcal{R}_{\mathcal{L}}$) for \mathcal{L} and is said to be a *writer* (*reader*). In the following discussion, which applies equally to read and write requests, we denote a request for \mathcal{L} of either kind as $\mathcal{X}_{\mathcal{L}}$.

$\mathcal{X}_{\mathcal{L}}$ is *satisfied* as soon as T_i^j holds \mathcal{L} , and *completes* when T_i^j releases \mathcal{L} . $\|\mathcal{X}_{\mathcal{L}}\|$ denotes the maximum time that T_i^j will hold \mathcal{L} . T_i^j becomes *blocked* on \mathcal{L} if $\mathcal{X}_{\mathcal{L}}$ cannot be satisfied immediately. (A resource can be held by multiple jobs simultaneously only if they are all readers.) If T_i^j issues another request \mathcal{X}' before \mathcal{X} is complete, then \mathcal{X}' is *nested* within \mathcal{X} . We assume without loss of generality that nesting is proper, *i.e.*, \mathcal{X}' must complete no later than \mathcal{X} completes. An *outermost* request is not nested within any other request.

When sharing resources in real-time systems, a locking protocol must be employed to both avoid deadlock and bound the maximum duration of blocking. In the following section, we present such a protocol.

²Tighter bounds are known for certain schedulers such as P-SP; see for example (Liu, 2000).

3 Reader-Writer Synchronization

The *flexible multiprocessor locking protocol* (FMLP) is a real-time mutex protocol based on the principles of flexibility and simplicity (Block et al., 2007) that has been shown to compare favorably to previously-proposed protocols (Block et al., 2007; Brandenburg and Anderson, 2008). In this section, we present an extended version of the FMLP with support for RW synchronization. We begin by giving a high-level overview of its core design.

The FMLP is considered to be “flexible” for two reasons: it can be used under G-EDF, P-EDF, as well as P-SP scheduling, and it is agnostic regarding whether blocking is implemented via spinning or suspending. Regarding the latter, resources are categorized as either “short” or “long.” Short resources are accessed using fair spin locks and long resources are accessed via a semaphore protocol. Whether a resource should be considered short or long is user-defined, but requests for long resources may not be contained within requests for short resources.

The terms “short” and “long” arise because (intuitively) spinning is appropriate only for very short critical sections, since spinning wastes processor time. However, two recent studies pertaining to shared in-memory data structures have shown that, in terms of schedulability, spinning is usually preferable to suspending when overheads are considered (Brandenburg and Anderson, 2008; Brandenburg et al., 2008b).³ Based on these trends, we restrict our focus to short resources in this paper and delegate the analysis of RW synchronization for long resources to future work.

3.1 Reader-Writer Request Rules

The reader-writer FMLP (RW-FMLP) is realized by the following rules, which are based on those given in (Block et al., 2007).

Resource groups. Nesting, which is required to cause a deadlock, tends to occur somewhat infrequently in practice (Brandenburg and Anderson, 2007). The FMLP strikes a balance between supporting nesting and optimizing for the common case (no nesting) by organizing resources into *resource groups*, which are sets of resources that may be requested together. Two resources are in the same group iff there exists a job that requests both resources at the same time. We let $\text{grp}(\mathcal{L})$ denote the group that contains \mathcal{L} . Deadlock is avoided by protecting each group by a *group lock*; before a job can access a resource, it must first acquire its corresponding group lock.⁴

Nesting. Read requests may be freely nested within other read and write requests, but write requests may only be nested within other write requests. We do not permit the nesting of write

³The studies presented in (Brandenburg and Anderson, 2008; Brandenburg et al., 2008b) considered request lengths in the range of a few microseconds, corresponding to request lengths that were measured in real systems (Brandenburg and Anderson, 2007). Suspension-based synchronization is likely preferable if critical sections are inherently long (*e.g.*, when synchronizing access to physical devices such as disks).

⁴Group-locking is admittedly a *very* simple deadlock avoidance mechanism that can be detrimental to throughput; however, the FMLP is the first multiprocessor real-time locking protocol that allows nesting of global resources at all. Obtaining a *provably* better protocol (in terms of worst-case blocking of outermost requests) remains an interesting open question.

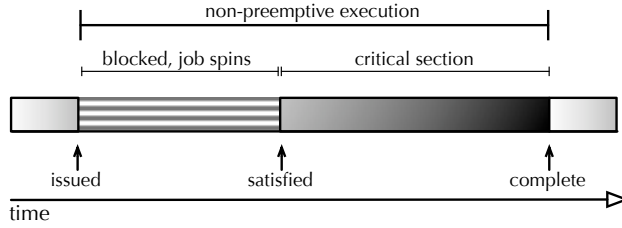


Figure 1: Illustration of an outermost request.

requests within read requests because this would require an “upgrade” to exclusive access, which is problematic for worst-case blocking analysis.

Requests. If a job T_i^j issues a read or write request $\mathcal{X}_{\mathcal{L}}$ for a short resource \mathcal{L} and $\mathcal{X}_{\mathcal{L}}$ is outermost, then T_i^j becomes non-preemptable and executes the corresponding entry protocol for $\text{grp}(\mathcal{L})$'s group lock (the details of which are discussed in the next section). $\mathcal{X}_{\mathcal{L}}$ is satisfied once T_i^j holds \mathcal{L} 's group lock. When $\mathcal{X}_{\mathcal{L}}$ completes, T_i^j releases the group lock and leaves its non-preemptive section. The execution of an outermost request is illustrated in Figure 1. If $\mathcal{X}_{\mathcal{L}}$ is not outermost, then it is satisfied immediately: if the outer request is a write request, then T_i^j already has exclusive access to the group, and if the outer request is a read request, then $\mathcal{X}_{\mathcal{L}}$ must also be a read request according to the nesting rule. In either case, it is safe to acquire \mathcal{L} .

The RW-FMLP can be integrated with the regular FMLP by replacing the FMLP's short request rules with the above-provided rules; regular short FMLP requests are then treated as short write requests.

3.2 Group Lock Choices

Group locks are the fundamental unit of locking in the FMLP and thus determine its worst-case blocking behavior. In this section, we consider four group lock choices (one of them a new RW lock) and discuss examples that illustrate how the use of RW locks and relaxed ordering constraints can significantly reduce worst-case blocking. Formal bounds on worst-case blocking are derived subsequently in Section A.

Task-fair mutex locks are considered here even though they are clearly undesirable for RW synchronization since they are the only spin-based locks for which bounds on worst-case blocking were derived in prior work. Hence, they serve in our experiments (see Section 5) as a performance baseline.

3.2.1 Task-fair Mutex Locks

With task-fair (or FIFO) locks, competing tasks are served strictly in the order that they issue requests. Task-fair mutex locks were employed in previous work on real-time synchronization because they have two desirable properties: first, they can be implemented efficiently (Mellor-Crummey and Scott, 1991a), and second, they offer strong progress guarantees. Since requests are executed non-preemptively (under the FMLP), task-fair locks ensure that a request of a job

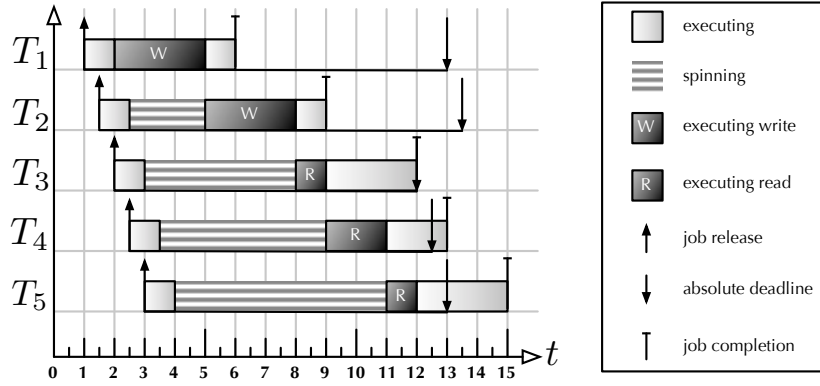


Figure 2: Example schedules of two writers (T_1, T_2) and three readers (T_3, T_4, T_5) sharing a resource \mathcal{L} that is protected by a task-fair mutex lock (tasks assigned to different processors). The jobs of T_4 and T_5 miss their respective deadlines at times 12.5 and 13.0 due to the unnecessary sequencing of readers.

is blocked once by at most $m - 1$ other jobs irrespective of the number of competing requests, which may be desirable as this tends to distribute blocking evenly among tasks.

However, enforcing strict mutual exclusion among readers is unnecessarily restrictive and can cause deadline misses. An example is shown in Figure 2, which depicts jobs of five tasks (two writers, three readers) competing for a resource \mathcal{L} . As \mathcal{L} 's group lock is a task-fair mutex lock, all requests are satisfied sequentially in the order that they are issued. This unnecessarily delays both T_4 and T_5 and causes them to miss their respective deadlines at times 12.5 and 13.

3.2.2 Task-fair RW Locks

With task-fair RW locks, while requests are still satisfied in strict FIFO order, the mutual exclusion requirement is relaxed so that the lock can be held concurrently by *consecutive* readers. This can lead to reduced blocking, as shown in Figure 3(a), which depicts the same arrival sequence as Figure 2. Note that the read requests of T_3, T_4 , and T_5 are satisfied simultaneously at time 8, which in turn allows T_4 and T_5 to meet their deadlines.

Unfortunately, task-fair RW locks may degrade to mutex-like performance when faced with a pathological request sequence, as is shown in Figure 3(b). The only difference in task behavior between Figure 3(a) and Figure 3(b) is that the arrival times of the jobs of T_1 and T_4 have been switched. This causes \mathcal{L} to be requested first by a reader (T_4 at time 2), then by a writer (T_2 at time 2.5), then by a reader again (T_3 at time 3), then by another writer (T_1 at time 3.5), and finally by the last reader (T_5 at time 4). Reader parallelism is eliminated in this scenario and T_5 misses its deadline at time 13 as a result.

3.2.3 Preference RW Locks

In a reader preference lock, readers are statically prioritized over writers, *i.e.*, writers are starved as long as there are unsatisfied read requests (Courtois et al., 1971; Mellor-Crummey and Scott, 1991b). The lack of strong progress guarantees for writers makes reader preference locks a

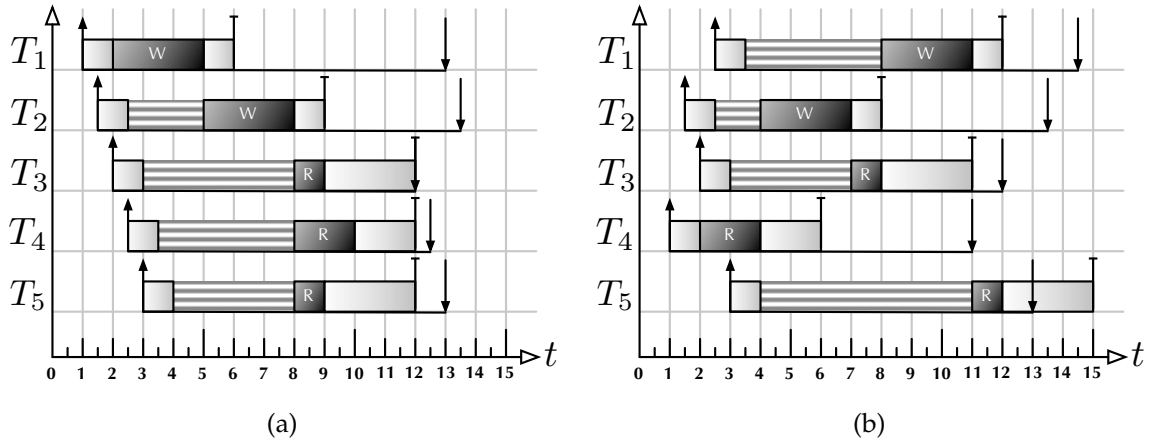


Figure 3: Example of reader parallelism (and lack thereof) under task-fair RW locks. **(a)** Given the arrival sequence depicted in Figure 2, all readers gain access in parallel and hence meet their respective deadlines. **(b)** If the arrival times of the jobs of T_1 and T_4 in Figure 3(a) are switched, then all readers are serialized and a deadline is missed. This demonstrates that task-fair RW locks are subject to mutex-like worst-case performance in the presence of multiple writers. (See Figure 2 for a legend.)

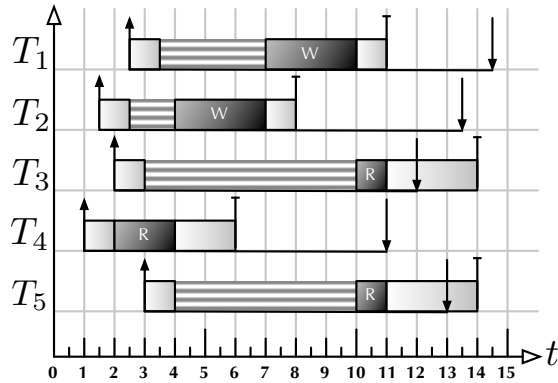


Figure 4: Example of reader delay under writer preference locks. Given the arrival sequence from Figure 3(b), two readers miss their respective deadlines (instead of only one as in Figure 3(b)) because all read requests remain unsatisfied while writers are present, which can significantly delay readers. (See Figure 2 for a legend.)

problematic choice for real-time systems because deadlines can be missed due to the resulting starvation.

Writer preference locks are an ill choice for similar reasons. This is illustrated in Figure 4, which depicts the same arrival sequence as Figure 3(b) but assumes that \mathcal{L} is protected by a writer preference lock. Again, \mathcal{L} is first acquired by a reader (T_4 at time 2) as there are initially no competing write requests. However, all reads are blocked as soon as writers are present (starting at time 2.5). Hence, T_3 and T_5 's read requests remain unsatisfied until both T_1 and T_2 's write requests have completed at time 10, which causes both readers to miss their respective deadline.

All previously-proposed RW locks fall within one of the categories discussed so far (or have no progress guarantees at all). Thus, the examples discussed above demonstrate that *no* such lock reliably reduces worst-case blocking (unless writers execute very infrequently).

3.2.4 Phase-fair RW Locks

The above examples reveal two root problems of existing RW locks: first, preference locks cause extended blocking due to intervals in which only requests of one kind are satisfied; and second, task-fair RW locks cause extended blocking due to a lack of parallelism when readers are interleaved with writers. A RW lock better suited for real-time systems should avoid these two pitfalls. These two requirements are captured by the concept of “phase-fairness.”

Definition 1. *A RW lock is phase-fair iff it has the following properties:*

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 read requests 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.*

Properties PF1 and PF3 ensure that a read request is never blocked by more than one writer phase and one reader phase *irrespective of the number of processors and the length of the write request queue*. Property PF4 ensures that reader phases end. Properties PF1 and PF2 ensure that a write request is never blocked by more than $m - 1$ reader and writer phases each (see Section 3.3 below).

In some sense, phase-fair locks can be understood as being a reader preference lock when held by a writer, and being a writer preference lock when held by readers, *i.e.*, readers and writers are “polite” and yield to each other.

Figure 5 depicts a schedule for the pathological arrival sequence from Figures 3(b) and 4 assuming a phase-fair group lock. T_4 issues the first read request and thus starts a new reader phase at time 2. T_2 issues a write request that cannot be satisfied immediately at time 2.5. However, T_2 's unsatisfied request prevents the next read request (issued by T_3 at time 3) from being satisfied due to Property PF4. At time 3.5, T_1 issues a second write request, and at time 4, T_5 issues a final read request. At the same time, T_4 's request completes and the first reader phase ends. The first writer phase lasts from time 4 to time 7 when T_2 's write request, which was first in the writer FIFO queue, completes. Due to Property PF1, this starts the next reader phase, and due to Property PF3, *all* unsatisfied read requests are satisfied. Note that, when T_5 's read request was issued, two write requests were unsatisfied. However, due to the phase-fair bound on read-request blocking, it was only blocked by one writer phase regardless of the arrival pattern. This allows all jobs to meet their deadlines in this example, and, in fact, for any arrival pattern of the jobs depicted in Figures 2–5.

This example suggests that phase-fair locks may be a desirable choice for real-time RW synchronization. However, in order to employ a locking protocol in real-time systems (and to

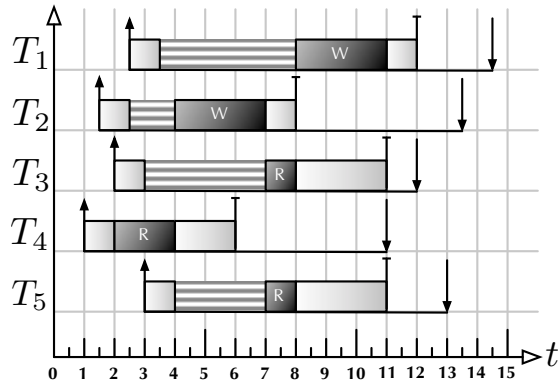


Figure 5: Example of reduced blocking under phase-fair locks. All readers meet their respective deadlines because they are not blocked by more than one write request (due to Property PF1). Note how the contending readers (T_3, T_4) yield to a writer (T_2) at time 4, which is followed in turn by a writer (T_1) yielding to the aforementioned readers: this exemplifies the “after-you politeness” implicitly required by the definition of phase-fairness. (See Figure 2 for a legend.)

properly compare locking choices), formal bounds on worst-case synchronization delays are required. We provide an overview of such bounds and summarize their key properties next.

3.3 Blocking Overview

Blocking is any delay encountered by a job that would not have arisen if all tasks were independent. A task’s maximum blocking duration must be bounded and accounted for when testing whether a task system is schedulable (Rajkumar, 1991).

There are two kinds of blocking that we must consider under the RW-FMLP: *direct blocking* and *arrival blocking*. A job T_i^j incurs direct blocking when it issues a request \mathcal{X} for a shared resource and \mathcal{X} is not immediately satisfied, which under the RW-FMLP happens only if T_i^j must spin while acquiring the corresponding group lock. T_i^j incurs arrival blocking when it is released with a sufficiently-high priority to be scheduled immediately, but cannot be scheduled because another job is executing non-preemptively (*i.e.*, it is spinning or executing a request) on T_i^j ’s assigned processor.

The derivation of reasonably-tight bounds on worst-case blocking that are sufficiently flexible to allow the analysis of non-trivial task systems incurs some inherent complexity. We provide full proofs in Appendix A for the interested reader; however, such level of detail is not required for the rest of the paper. Instead, we provide a short summary that is sufficient to appreciate the key differences among the considered lock types. Note, however, that the following high-level discussion is not a substitute for the complete analysis presented Appendix A and should not be used for schedulability analysis—it merely highlights simplified instantiations of the derived bounds for particular scenarios to convey an intuitive understanding.

Table 1 depicts bounds on worst-case blocking under three scenarios, expressed in terms of the number of blocking phases. Let T_i^j denote the job under consideration.

The *Single Write* scenario assumes that T_i^j issues exactly one write and no read requests. Since writers require exclusive access, the task-fair mutex, task-fair RW, and writer preference

RW bounds reflect the fact that a blocking write request may be executed on every other processor. Under reader preference locks, write requests are subject to starvation and it is thus not possible to bound the number of blocking phases based purely on T_i^j 's requests— T_i^j will be blocked as long as other jobs keep issuing read requests. Finally, the phase-fair bound reveals the negative impact of enforcing alternating reader and writer phases: while T_i^j is blocked by at most $m - 1$ writer phases, it can also be *transitively* blocked by $m - 1$ interspersed reader phases.

The *Single Read* scenario assumes the complimentary situation, *i.e.*, T_i^j issues one read and no write request. This clearly reveals the problem with task-fair locks—under both task-fair mutex and RW locks, read requests may be delayed significantly while progressing through the FIFO queue. Starvation is again an issue under preference locks. However, the situation is reversed: writer preference locks allow reads to be delayed indefinitely, whereas reader preference locks offer the lowest bound of all locks for reader blocking—at most one writer phase can block T_i^j if said writer phase was active when T_i^j issued its read request. Phase-fair RW locks offer a close approximation of reader preference locks under this scenario, since satisfying all blocked read requests at the beginning of each reader phase (Property PF3) ensures that at most two phases block a read request. This shows that forcing phases to alternate (Property PF1) benefits readers at the expense of writers.

The third scenario, *Repeated Reads*, looks at cumulative blocking across multiple read requests over some interval $[t_0, t_1)$ in which T_i^j issues many read requests in quick succession.⁵ With regard to this interval, W denotes the total number of potentially-blocking write requests, *i.e.*, requests issued by jobs other than T_i^j , and R denotes the total number of potentially-blocking read requests. Further, it is assumed that $2W < R$, *i.e.*, that reads are much more frequent than writes. This scenario reveals why mutex locks are an undesirable choice for RW synchronization: in the worst case, T_i^j is delayed by every other request. In contrast, under task-fair RW locks, T_i^j is only blocked by at most twice the number of potentially-blocking write requests. Intuitively, this is because one of T_i^j 's read requests is only blocked by a reader phase if they are “separated” by a writer phase. Writer preference locks suffer from the same shortcoming as mutex locks—in the worst case, T_i^j is blocked by all other requests. Under reader preference locks, read requests are never blocked by reader phases—thus, at most W requests can block T_i^j in this case. Finally, phase-fair RW locks maintain the desirable bound of task-fair RW locks, since, under phase-fairness, reader phases only block read requests if a writer phase is blocking, too.

These comparisons substantiate the observations of Section 3.2. In particular,

- task-fair mutex and task-fair RW locks have similar worst-case blocking behavior for individual requests;
- preference locks offer appealing worst-case behavior only for the prioritized request type and give rise to starvation; and
- phase-fair RW locks strike a compromise between the desirable properties of reader preference and task-fair RW locks: reads are only blocked by a constant number of phases and writes are not starved, albeit at the cost of a factor of two in each bound (compared to the best bound in each scenario).

⁵ T_i^j is assumed to issue sufficiently many read requests to be blocked by all requests of other tasks that can block T_i^j , *i.e.*, per-request bounds are assumed to not affect the outcome.

Lock Type	Single Write	Single Read	Repeated Reads
task-fair mutex	$m - 1$	$m - 1$	$W + R$
task-fair RW	$m - 1$	$m - 1$	$2W$
writer pref. RW	$m - 1$	∞	$W + R$
reader pref. RW	∞	1	W
phase-fair RW	$2 \cdot (m - 1)$	2	$2W$

Table 1: A comparison of the bounds on worst-case direct blocking in three different scenarios under each of the considered group lock choices. (Recall that m denotes the number of processors.) The second column (*Single Write*) lists the maximum number of phases (either read or write) that can block a job if it issues exactly one write and no read requests. The third column (*Single Read*) lists the maximum number of phases (either read or write) that can block a job if it issues exactly one read and no write requests. The fourth column (*Multiple Reads*) lists the maximum number of phases that can block a job that repeatedly issues a read requests during some interval $[t_0, t_1)$, where W denotes the number of competing, *i.e.*, potentially blocking, write requests and R denotes the number of competing read requests issued during $[t_0, t_1)$ (assuming that $2W < R$). Note that phase-fair RW locks are the only lock type that offers both $O(1)$ read and $O(m)$ write blocking. These bounds are simplified special cases of those proved in Appendix A.

We expect the fact that the phase-fair bound on read request blocking is $O(1)$ —and not $O(m)$ —to become increasingly significant as multicore platforms become larger.

4 Implementing Phase-Fair Reader-Writer Locks

The preceding discussion indicates that phase-fairness can reduce worst-case blocking significantly. However, to be a viable choice, phase-fair RW locks must be efficiently implementable on common hardware platforms. Further, in practice, appropriate definitions of “efficient” may vary widely between applications.

One commonly-used complexity metric for locking algorithms is to count *remote memory references* (RMR) (Anderson et al., 2003), *i.e.*, on a cache-coherent multiprocessor, locks are classified by how many cache invalidations occur per request under maximum contention. The assumption underlying RMR complexity is that cache-local operations are (uniformly) cheap and that therefore the number of (uniformly) expensive cache invalidations dominate lock acquisition costs.

Cache consistency traffic can certainly severely limit performance, but, in practice, the efficiency of locks also strongly depends on both the number and cost of the required atomic operations. For example, on a given hypothetical platform, a lock implementation that requires multiple “light-weight” atomic-add instructions may outperform alternate implementations that rely on fewer “heavy-weight” compare-and-swap instructions. Such tradeoffs are strongly hardware dependent and can only be resolved by benchmarking actual lock performance on the platform(s) of interest; Section 5 further explores this issue.

A third efficiency criterion arises in the design of memory-constrained (embedded) systems,

wherein lock size concerns may outrank the desire for low acquisition overheads. Lock size can especially become a concern when locks are used to protect individual data objects (as opposed to locking code paths), which is often desirable if high throughput is required as fine-grained locking can increase parallelism.

In this section, we present three phase-fair RW lock implementations, each addressing one of the aforementioned efficiency requirements:

- a simple-to-implement *ticket-based* phase-fair RW lock, denoted PF-T, that only depends on hardware support for atomic-add, fetch-and-add, and atomic stores, and that requires only two atomic operations⁶ each on the reader and writer path;
- a *compact* version of the first algorithm for memory-constrained system, denoted PF-C, that requires only four instead of 16 bytes per lock, but at the price of requiring additional atomic operations; and
- a *queue-based* implementation, denoted PF-Q, that requires only a constant number of remote memory references.

These algorithms heavily reuse and extend Mellor-Crummey and Scott's *ticket* and *queue lock* techniques (Mellor-Crummey and Scott, 1991a,b).

4.1 A Simple Phase-Fair Reader-Writer Lock

The PF-T algorithm, as given in its entirety in Listing 1, assumes a 32-bit little-endian architecture. However, it can be easily adapted to other native word sizes and big-endian architectures. We discuss its structure and the entry and exit procedures, which are both illustrated in Figure 7, next.

Structure. The PF-T lock consists of four counters that keep track of the number of issued (*rin*, *win*) and completed (*rout*, *wout*) read and write requests (lines 1–3 of Listing 1). *rin* serves multiple purposes: bits 8–31 are used for counting issued read requests, while bit 1 (*PRES*) is used to signal the presence of unsatisfied write requests and bit 0 (*PHID*) is used to tell consecutive writer phases apart. For efficiency reasons (explained below), bits 2–7 remain unused, as do bits 0–7 of *rout* for reasons of symmetry. The allocation of bits in *rin* and *rout* is illustrated in Figure 6.

Readers. The reader entry procedure (lines 10–13, illustrated in Figure 7) works as follows. First, a reader atomically increments *rin* and observes *PRES* and *PHID* (line 12). If no writer is present ($w = 0$), then the reader is admitted immediately (line 13). Otherwise, the reader spins until either of the two writer bits changes: if both bits are cleared, then no writer is present any longer, otherwise—if only *PHID* toggles but *PRES* remains unchanged—the beginning of a reader phase has been signaled. The reader exit procedure (lines 15–16) only consists of atomically incrementing *rout*, which allows a blocked writer to detect when the lock ceases to be held by readers (as discussed below, line 24).

⁶Not counting atomic stores. Stores are atomic by default on many current platforms (e.g., Intel x86).

```

type pft_lock = record 1
  rin, rout : unsigned integer { initially 0 } 2
  win, wout : unsigned integer { initially 0 } 3

const RINC = 0x100 { reader increment } 5
const WBITS = 0x3 { writer bits in rin } 6
const PRES = 0x2 { writer present bit } 7
const PHID = 0x1 { phase id bit } 8

procedure read_lock(L : ^pft_lock) 10
  var w : unsigned integer; 11
  w := fetch_and_add(&L->rin, RINC) and WBITS; 12
  await (w = 0) or (w ≠ L->rin and WBITS) 13

procedure read_unlock(L : ^pft_lock) 15
  atomic_add(&L->rout, RINC) 16

procedure write_lock(L : ^pft_lock) 18
  var ticket, w : unsigned integer; 19
  ticket := fetch_and_add(&L->win, 1); 20
  await ticket = L->wout; 21
  w := PRES or (ticket and PHID); 22
  ticket := fetch_and_add(&L->rin, w); 23
  await ticket = L->rout 24

procedure write_unlock(L : ^pft_lock) 26
  var lsb : ^unsigned byte; 27
  lsb := &L->rin; 28
  { lsb^ = least-significant byte of L->rin } 29
  lsb^ := 0; 30
  L->wout := L->wout + 1 31

```

Listing 1: The ticket-based PF-T algorithm. This implementation assumes little-endian 32-bit words. A PF-T lock requires 16 bytes.

Writers. Similarly to Mellor-Crummey and Scott’s simple task-fair RW lock (Mellor-Crummey and Scott, 1991b), FIFO ordering of writers is realized with a ticket abstraction (Mellor-Crummey and Scott, 1991a). The writer entry procedure (lines 18–24) starts by incrementing *win* in line 20 and waiting for all prior writers to release the lock (line 21). Once a writer is the head of the writer queue (*ticket = wout*), it atomically sets *PRES* to one, sets *PHID* to equal the least-significant bit of its ticket, and observes the number of issued read requests (lines 22–23). Note that the least-significant byte of *rin* equals zero when observed in line 23 since no other writer can be present. Finally, the writer spins until all readers have released the lock before entering its critical section in line 24. The writer exit procedure consists of two steps. First, the beginning of a reader phase is signaled by clearing bits 0–7 of *rin* by atomically writing zero to its least-significant byte (lines 28–30). Clearing the complete least-significant byte instead of just bits 0 and 1 is a performance optimization since writing a byte is usually much faster than atomic read-modify-write instructions on modern hardware architectures. Finally, the writer queue is updated by incrementing *wout* in line 31.

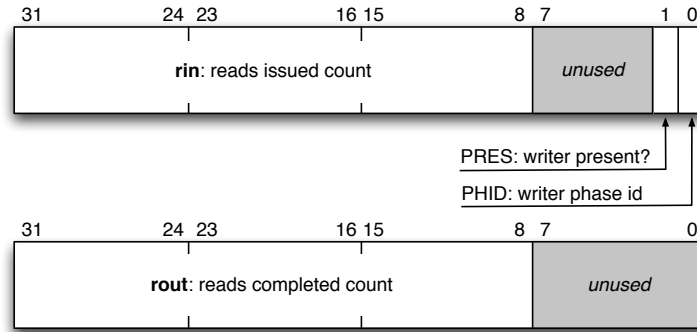


Figure 6: The allocation of bits in the reader entry counter and the reader exit counter of the PF-T algorithm (corresponding to line 2 of Listing 1).

Phase id bit. The purpose of *PHID* is to avoid a potential race between a slow reader (\mathcal{R}_1) and two writers ($\mathcal{W}_1, \mathcal{W}_2$). Assume that \mathcal{W}_1 holds the lock and that \mathcal{R}_1 and \mathcal{W}_2 are blocked. When \mathcal{W}_1 releases the lock, *PRES* is cleared (line 30) and *wout* is incremented (line 31). Subsequently, \mathcal{W}_2 re-sets *PRES* (line 23) and waits for \mathcal{R}_1 to release the lock. In the absence of *PHID*, \mathcal{R}_1 could fail to observe the short window during which *PRES* is cleared and continue to spin in line 13, waiting for the *next* writer phase to end. This deadlock between \mathcal{R}_1 and \mathcal{W}_2 is avoided by checking *PHID*: if \mathcal{R}_1 misses the window between writers, it can still reliably detect the beginning of a reader phase when *PHID* is toggled.

Correctness. If there are at most $2^{32} - 1$ concurrent writers and at most $2^{24} - 1$ concurrent readers, then the PF-T lock ensures exclusion of readers and writers and mutual exclusion among writers.⁷ Overflowing *rin*, *win*, *rout*, and *wout* is harmless because the counters are only tested for equality: in order for two writers to enter their critical sections concurrently, they must obtain the same ticket value. This is only possible if *win* wrapped around, *i.e.*, if more than $2^{32} - 1$ writers draw a ticket in between times that *wout* is incremented. Similarly, a writer can only enter its critical section during a reader phase if there are at least 2^{24} reads in progress at the time that the writer set *PRES*. In order for a reader to enter during a writer phase, either *PHID* would have to toggle or *PRES* would have to clear, which is only possible if a writer becomes head of the writer queue while another writer is still executing its critical section—which, as argued above, is impossible.

Liveness is guaranteed because all readers will eventually observe that the *PHID* bit was toggled, and because a blocked writer will eventually observe that *rout* increased. PF-T locks are phase-fair because readers cannot enter while a writer is spinning (as *PRES* is set in this case), and because writers unblock all waiting readers both as part of the writer exit procedure (by clearing *PRES*) and the writer entry procedure (by toggling *PHID*).

⁷Note that there can be at most m concurrent readers and writers under the FMLP since requests are executed non-preemptively.

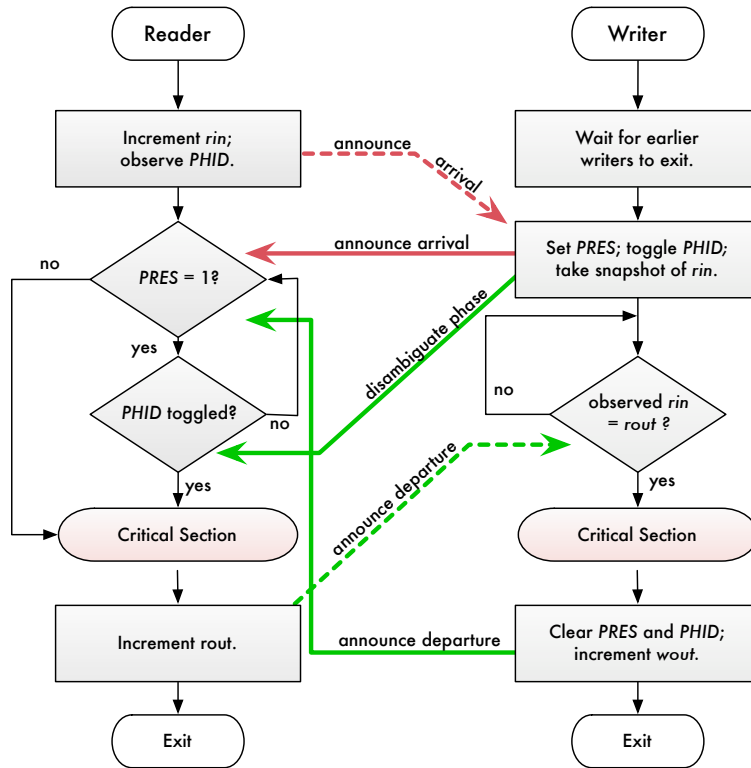


Figure 7: An illustration of the reader and writer control flow in the PF-T algorithm. Directed edges indicate control flow; bold arrows indicate communication, where solid arrows indicate information flow from writers to readers and dashed arrows indicate information flow from readers to writers. The linearization point for concurrent reader and writer arrivals is the update of rin —if the reader increments rin before the writer sets $PRES$, then the reader enters first, otherwise, if the writer first sets $PRES$ before rin is incremented, then the writer precedes the reader. (Recall from Figure 6 that the $PRES$ bit is part of the rin word.) Deadlock between a slow reader and a quickly-arriving writer is avoided by toggling $PHID$ when setting $PRES$.

4.2 A Compact Phase-Fair Reader-Writer Lock

In Listing 1, rin , win , $rout$, and $wout$ are each defined as four-byte integers. However, requiring 16 bytes per lock may be excessive in the context of memory-constrained applications (e.g., embedded systems). To accommodate such constraints, we introduce the PF-C algorithm that only requires 4 bytes and supports up to 127 concurrent readers and writers next. The PF-C algorithm, which is given in its entirety in Listing 2, closely resembles the PF-T lock given in Listing 1; we focus on the notable differences in the following discussion. Except for minor details, the illustration of PF-T’s control flow (Figure 7) applies to PF-T as well.

Structure. The lock itself consists of only one 32-bit word (line 1). The four ticket counters rin , $rout$, win , and $wout$ are collapsed into four 7-bit-wide bit fields, as shown in Figure 8. The four counters are separated by one bit each that serves as an *overflow guard*. Initialized to zero, an overflow guard prevents an overflow in one counter from corrupting the adjacent counter—of course, this requires the overflow guard to be reset before a second overflow can occur (as

```

type pfc_lock = unsigned integer { initially 0 } 1

const WOUT_SHIFT = 1 { least-significant bit of wout } 3
const WIN_SHIFT = 9 { least-significant bit of win } 4
const RIN_SHIFT = 17 { least-significant bit of rin } 5
const ROUT_SHIFT = 25 { least-significant bit of rout } 6
const MASK = 0x7f { mask for rin/win/rout/wout } 7
const GUARD = 0x80 { mask for the overflow guard bit } 8
const PRES = 0x1 { mask for PRES } 9
const WBITS = 0x3 { mask for PRES and PHID } 10

procedure read_lock(L : ^pfc_lock) 12
  var ticket, w : unsigned integer; 13
  ticket := fetch_and_add(L, 1 shl RIN_SHIFT); 14
  if ((ticket shr RIN_SHIFT) and MASK) = MASK then 15
    atomic_sub(L, GUARD shl RIN_SHIFT) 16
  end if 17
  w := ticket and WBITS; 18
  await ((w and PRES) = 0) or (w ≠ (L^ and WBITS)) 19

procedure read_unlock(L : ^pfc_lock) 21
  atomic_add(L, 1 shl ROUT_SHIFT) 22

procedure write_lock(L : ^pfc_lock) 24
  var ticket : unsigned integer; 25
  ticket := fetch_and_add(L, 1 shl WIN_SHIFT); 26
  ticket := (ticket shr WIN_SHIFT) and MASK; 27
  if ticket = MASK then 28
    atomic_sub(L, GUARD shl WIN_SHIFT) 29
  end if 30
  await ticket = ((L^ shr WOUT_SHIFT) and MASK); 31
  ticket := fetch_and_add(L, 1); 32
  ticket := (ticket shr RIN_SHIFT) and MASK; 33
  await ticket = ((L^ shr ROUT_SHIFT) and MASK) 34

procedure write_unlock(L : ^pfc_lock) 36
  if ((L^ shr WOUT_SHIFT) and MASK) = MASK then 37
    atomic_sub(L, (GUARD shl WOUT_SHIFT) - 1) 38
  else 39
    atomic_add(L, 1) 40
  end if 41

```

Listing 2: The ticket-based PF-C algorithm. This implementation assumes little-endian 32-bit words. A PF-C lock requires only 4 bytes.

discussed below).

The ticket counters are incremented with fetch-and-add, atomic-add, and atomic-sub⁸ operations in which the second argument, *i.e.*, the value to be added/subtracted, has been shifted by the offset of the to-be-updated ticket counter. The respective offsets are given in lines 3-6 of

⁸An atomic-sub instruction can be trivially emulated with an atomic-add instruction, thus the PF-C lock does not require additional hardware support (compared to the requirements of the PF-T algorithm).

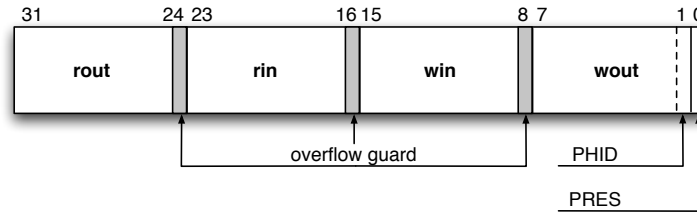


Figure 8: The allocation of bits in a PF-C lock (corresponding to line 1 of Listing 2). The four ticket counters *rin*, *rout*, *win*, and *wout* consist of only seven bits each and are separated by overflow guards, *i.e.*, bits that are otherwise unused and only employed to detect when each counter wraps. Due to the limited size of the ticket counters, at most 127 readers and 127 writers may issue requests concurrently. Note that *PHID* overlaps with the least-significant bit of *wout*. (*PRES* does not overlap with any counter.)

Listing 2.

Similarly, the ticket counters are observed by reading the lock word and then shifting and masking the observed value. The bit mask *MASK* (line 7 in Listing 2) corresponds to the width of one counter (seven bits).

Compared to the PF-T bit allocation, the positions of the phase id bit, *PHID*, and the writer present bit, *PRES*, are reversed in order to overlay *PHID* with the least-significant bit of *wout*—since *wout* is only incremented at the end of each writer phase, the toggling of the least-significant bit of *wout* implies the beginning of the next reader phase (if any readers are present). Thus, it is unnecessary to allocate a bit exclusively for *PHID*.

Readers. The PF-C reader entry procedure (lines 12–19) implements the same steps as the PF-T reader entry procedure: a reader T_i^j first atomically increases *rin* by one (subject to shifting, line 13) and observes both writer bits (line 17), and then—if a writer is present—spins until a writer bit toggles (line 18).

However, an additional step is required to avoid corrupting *rout* when *rin* overflows. If all bits of *rin* were set *before* it was incremented, *i.e.*, if all *rin* bits are set in T_i^j 's ticket (line 15), then T_i^j caused *rin* to overflow. Thus, *rin*'s overflow guard bit is set and must be reset before *rin* overflows again. Clearing the overflow bit is accomplished by atomically subtracting *GUARD* (after shifting it to *rin*'s position) from the lock word (line 16). As long as there are at most 127 concurrent readers, *rin* cannot overflow a second time before T_i^j has cleared the overflow guard, thus this mechanism ensures that *rin* never corrupts *rout*.

The reader exit procedure is trivial: *rout* is simply incremented (lines 21–22). Since *rout* is the left-most field, it can safely overflow without corrupting any other field. This allocation is intentional—it enables a branch-free and short (and thus fast) reader exit path, which is likely executed more frequently than the writer exit path.

Writers. In the first part of the writer entry procedure, a newly arriving writer increments *win* (subject to shifting, line 26), extracts its ticket from the observed lock value (by shifting, line 27), checks for and corrects the overflow (if it occurred, lines 28–30), and then spins until all previous writes have completed (line 31).

Once a writer has become the head of the writer queue, it blocks newly arriving readers from entering their critical sections by setting *PRES*—since *PRES* is the first bit in the lock word, this is simply done by atomically adding one to the lock value (line 32). The writer then waits for all readers to exit by waiting for *rout* to equal *rin*'s observed value at the time of setting *PRES* (lines 33–34).

Conceptually, the writer exit procedure (lines 36–41) has to accomplish three goals: *PRES* must be cleared, *wout* must be incremented, and overflowing *wout* must be avoided (or corrected). Since *wout* is allocated directly after *PRES*, it is possible to combine all three into one atomic update. Note that *wout* is never updated concurrently,⁹ hence a possible overflow can be detected before it is updated (lines 37). First consider the case that *wout* does not overflow (*i.e.*, the else-branch in lines 39–41). In this case, it is sufficient to clear *PRES* and increment *wout*. Since *PRES*, which is the least-significant bit of the lock word, is known to be set, both can be accomplished by adding one to the lock word—the addition will carry into *wout* and *PRES* will be cleared.

If incrementing *wout* does cause an overflow, then *wout*'s overflow guard must be cleared by subtracting *GUARD* (shifted to *wout*'s position) after updating *PRES* and *wout*. Since $1 - \text{GUARD} = -(\text{GUARD} - 1)$, both updates can be combined into one atomic subtraction (line 38). The overflow guard between *win* and *wout* is thus unused, but cannot be allocated to either *win* or *wout* since they must be of equal size.

Correctness. Similar to PF-T locks, since all counters are seven bit wide (and possible overflow is handled in both reader and writer paths), PF-C locks are correct if there are at most $2^8 - 1$ concurrent readers and writers each.¹⁰ PF-C locks implement the same control flow as PF-T locks and hence are also phase-fair.

4.3 A Queue-Based Phase-Fair Reader-Writer Lock

Mellor-Crummey and Scott showed in their seminal work on efficient spin-based synchronization that it is possible to implement both task-fair mutex and RW locks as *queue locks* such that spinning causes only a constant number of cache invalidations, *i.e.*, with $O(1)$ RMR complexity, irrespective of the number of processors (Mellor-Crummey and Scott, 1991a,b).

In a queue lock, a blocking job enqueues a *queue node* in a linked list prior to spinning on a flag in the queue node. Thus, since each spinning job spins on a private location, it will only incur a cache invalidation when it is unblocked.¹¹

In contrast, under the PF-T algorithm, a spinning reader may incur up to m cache invalidations since subsequently arriving readers atomically update *rin*, which causes the cache line holding *rin* to be evicted from the caches of all spinning readers. Similarly, a writer may incur up to m cache invalidations in total while spinning either on *wout* or *rout*. Further, under the PF-C algorithm, all readers and writers modify the same word, hence a spinning job may

⁹However, other bitfields in the lock word may still be updated concurrently due to reader and writer arrivals, thus—in contrast to the PF-T algorithm—all updates must be atomic.

¹⁰Note that bits can be re-allocated from *win* and *wout* to *rin* and *rout* to enable higher reader counters, albeit at the expense of reduced writer counts.

¹¹We assume familiarity with queue lock implementation techniques—see (Anderson et al., 2003) for a survey.

incur up to $2m$ cache invalidations.¹² Thus, both PF-T and PF-C locks are only of $O(m)$ RMR complexity.

We show how spin queues can be applied to reduce RMR complexity under phase-fair locks next. The resulting PF-Q algorithm is given in Listings 3 and 4.

Structure. From a high level, the PF-Q algorithm resembles the PF-T algorithm: readers are tracked by the *rin* and *rou*t counters (line 14 in Listing 3), and are blocked from entering their critical sections by setting the *PRES* bit in *rin* (line 3). The key difference is that instead of spinning on the lock state itself, blocked jobs enqueue themselves into reader and writer queues.

There are three such queues: a writer queue and two reader queues. The pointer *wtail* (line 17) holds the address of the last writer’s queue node, or has the value *NIL* (line 6) if no writer is present. The pointer *whead* (line 17) holds the address of the head of the writer queue’s node if said writer is blocked by a reader phase. There are two reader queues, pointed to by *rtail*[0] and *rtail*[1] (line 16), corresponding to the two possible values of the phase id bit in *rin* (line 2). Each *rtail* pointer can assume two special values: *NIL*, which signals that arriving readers may proceed, and *WAIT* (line 7), which signals that arriving readers should spin.

A notable difference to PF-T locks is that the *PRES* bit is set by writers in both *rin* and *rou*t, with differing semantics. If *PRES* is set in *rin*, then readers are not allowed to enter their critical sections and must spin. If *PRES* is set in *rou*t, then a writer is awaiting the end of a reader phase and the last exiting reader must unblock the spinning writer. Since readers may leave their critical sections in an order different from the arrival sequence, the last reader can only be identified when it updates *rou*t. For this purpose, the variable *last* holds the value that *rou*t will assume once the reader phase has ended (as discussed below).

Readers. Arriving readers start by making their presence known and observe *PRES* by incrementing *rin* (line 22). If *PRES* is set, then the reader must block (line 23). To do so, it determines the current phase id (either zero or one, line 24), initializes its queue node (line 25), and appends its queue node onto the end of the reader queue corresponding to the current phase by atomically exchanging the tail pointer (line 26). The queue update yields *prev*, which is either the address of the predecessor’s queue node, *WAIT* if the queue was empty, or *NIL* if the blocking writer phase ended in the mean time, *i.e.*, after *PRES* was observed.

In the non-*NIL* case (lines 28–31), the reader spins on its private *blocked* flag (line 28)—the reader has made its presence known, and thus can simply wait until it is notified of the end of the blocking writer phase. Once unblocked, it checks whether it is the head of the reader queue (line 29)—if not, it notifies its predecessor by clearing the corresponding *blocked* flag before entering its critical section. Note that, in contrast to the task-fair queue lock presented in (Mellor-Crummey and Scott, 1991b), readers are unblocked from tail to head.

If *prev* equals *NIL* (lines 32–36), then the arriving reader raced with an exiting writer and observed *rin* before *PRES* was cleared. In this case, the reader must proceed to enter its critical section since the writer has already exited. However, additional readers may have observed *PRES*, enqueued themselves onto the reader queue, and entered the non-*NIL* case—and thus

¹²As modern architectures with cache line sizes of less than 16 bytes are rare, a spinning job may likely incur up to $2m$ cache invalidations under PF-T locks in practice, too.

```

const RINC    =    0x100 { reader increment } 1
const PHID    =    0x1  { phase id bit } 2
const PRES    =    0x2  { writer present bit } 3
const WMSK    =    0x3  { writer bits in rin/rout } 4
const TMSK    = not WMSK { reader ticket bits in rin/rout } 5
const NIL    { NIL and WAIT are two address-like values } 6
const WAIT    { that are distinct from any legal address } 7

type pfq_node = record 9
  next      : ^wnode      { initially 0 } 10
  blocked   : boolean    { initially false } 11

type pfq_lock = record 13
  rin, rout : unsigned integer { initially 0 } 14
  last      : unsigned integer { initially 0 } 15
  rtail     : array [0..1] of ^pfq_node { initially [NIL, NIL] } 16
  wtail, whead : ^pfq_node { initially NIL } 17

procedure start_read(L : ^pfq_lock; I : ^pfq_node) 19
  var ticket, phase : unsigned integer; 20
  var prev          : ^pfq_node; 21
  ticket := fetch_and_add(&L->rin, RINC); 22
  if (ticket and PRES) = PRES then 23
    phase := ticket and PHID; 24
    I->blocked := true; 25
    prev := fetch_and_store(&L->rtail[phase], I); 26
    if prev ≠ NIL then 27
      await not I->blocked; 28
      if prev ≠ WAIT then 29
        prev->blocked := false 30
      end if 31
    else 32
      prev := fetch_and_store(&L->rtail[phase], NIL); 33
      prev->blocked := false; 34
      await not I->blocked 35
    end if 36
  end if 37

procedure end_read(L : ^pfq_lock; I : ^pfq_node) 39
  var ticket : unsigned integer; 40
  ticket := fetch_and_add(&L->rout, RINC); 41
  if (ticket and PRES) = PRES and 42
    (ticket and TMSK) = L->last - 1 then 43
    L->whead->blocked := false 44
  end if 45

```

Listing 3: The queue-based PF-Q algorithm—type declarations and reader entry and exit procedures.

may be waiting to be unblocked (line 28). To avoid deadlock in this case, the reader that observed (and replaced) *NIL* must restore *rtail[phase]* (line 33) and unblock the job at the tail of the reader queue (line 34). Note that the waiting readers propagate the update to the head of

```

procedure start_write(L : ^pfq_lock; I : ^pfq_node)           46
  var phase, ticket, exit : unsigned integer                 47
  var prev : ^pfq_node;                                       48
  prev := fetch_and_store(&L->wtail, I);                       49
  if prev ≠ NIL then                                         50
    I->blocked := true;                                       51
    prev->next := I;                                           52
    await not I->blocked;                                       53
  end if                                                       54
  I->blocked := true;                                       55
  L->whead := I;                                              56
  phase := L->rin and PHID;                                    57
  L->rtail[phase] := WAIT;                                     58
  ticket := fetch_and_add(&L->rin, PRES);                     59
  ticket := ticket and TMSK;                                  60
  L->last := ticket;                                          61
  exit := fetch_and_add(&L->rout, PRES);                       62
  exit := exit and TMSK;                                      63
  if exit ≠ ticket then                                       64
    await not I->blocked;                                       65
  end if                                                       66

procedure end_write(L : ^pfq_lock; I : ^pfq_node)           68
  var phase : unsigned integer;                               69
  var lsb : ^unsigned byte;                                    70
  var prev : ^pfq_node;                                       71
  L->rout := L->rout and TMSK;                                  72
  phase := L->rin and PHID;                                    73
  lsb := &L->rin;                                              74
  { lsb^ = least-significant byte of L->rin }                 75
  lsb^ := (phase + 1) and PHID;                               76
  prev := fetch_and_store(&L->rtail[phase], NIL);            77
  if prev ≠ WAIT then                                       78
    prev->blocked := false;                                    79
  end if                                                       80
  if I->next ≠ NIL or not                                       81
    compare_and_swap(&L->wtail, I, NIL) then                 82
      await I->next ≠ NIL;                                     83
    end if                                                       84
  if I->next ≠ NIL then                                       85
    I->next->blocked := false;                                  86
  end if                                                       87

```

Listing 4: The queue-based PF-Q algorithm—writer entry and exit procedures.

the queue. Note also that the head of the queue is the job that observed *NIL*, thus it is already aware of the end of the blocking writer phase. However, the head must still wait for its *blocked* flag to be toggled before entering its critical section (line 35) because otherwise its successor, if sufficiently delayed, could access *I* after it has been deallocated.¹³ (If there are no successors, then *prev* = *I*, and thus *blocked* = *false* due to the update in line 34.)

¹³Queue nodes are commonly allocated on the stack, hence delayed writes must be avoided.

As in the previous two phase-fair locks, exiting readers increment the *rou*t counter (line 41). In the PF-T and PF-C locks, the writer spins by comparing *wout* with its snapshot of *rin*—this, of course, can cause repeated cache invalidations. Thus, a different approach for detecting the end of a reader phase is required: if the *PRES* bit is set in *rou*t (line 42), then the head of the writer queue is waiting to be unblocked by the last exiting reader (and the *whead* pointer is valid). To reliably identify the last reader, the writer records its snapshot of *rin* in *last*. Thus, each exiting reader can test whether it is the last reader (line 43),¹⁴ and if so, unblock the waiting writer (line 44).

Writers. Lines 49–55 of the writer entry procedure and lines 81–87 of the writer exit procedure incorporate Mellor-Crummey and Scott’s mutex queue lock (Mellor-Crummey and Scott, 1991a) to ensure FIFO queueing of writers: an arriving writer appends its queue node to the writer queue (line 49), updates its predecessor’s *next* pointer (if any, lines 50–52), and then spins until unblocked (line 51). Symmetrically, an exiting writer removes itself from the writer queue (lines 81–84) and unblocks its successor (line 85–87).

Once a writer is at the head of the writer queue (line 55), it stores the address of its queue node in *whead* (line 56), determines the phase id (line 57), and initializes the corresponding reader queue (line 58). Then it stops newly-arriving readers from entering their critical sections by setting *PRES* in *rin* (line 59). The observed value of *rin* is stored in *last* to make it available to readers (as discussed above, line 61). Then it sets *PRES* in *rou*t to make the presence of a blocked writer known to exiting readers. While setting *PRES*, the writer also observes *rou*t—blocking is only necessary if *rou*t is not equal the previously-observed value of *rin*, otherwise all earlier-arrived readers have already finished their respective critical sections (lines 64–65).

An exiting writer first clears the *PRES* bit in *rou*t (line 72), which can be done non-atomically since no concurrent reader exists. Next, it initiates the next reader phase. First, it enables newly-arriving readers to enter their critical sections by clearing the *PRES* bit in *rin* (lines 73–76). This is accomplished by writing the next phase id to the least-significant byte of *rin* to avoid the use of an atomic fetch-and-modify instruction (*i.e.*, *PHID* is toggled, line 76). Finally, any spinning readers are unblocked by storing *NIL* in the *rtail* pointer corresponding to the ending writer phase (line 77) and clearing the *blocked* flag of the tail node (if any, lines 78–79).

Correctness. Mutual exclusion of writers results from the use of Mellor-Crummey and Scott’s task-fair mutex queue lock. Readers cannot enter during a writer phase since *PRES* is not cleared until the writer exits. Liveness for writers is ensured because they atomically check *rou*t and make their presence known to readers. Liveness for readers is ensured because the use of two special values (*NIL* and *WAIT*) allows races between exiting writers and entering readers to be detected and corrected. Readers that are still in the process of propagating the start of a new reader phase through the reader queue do not interfere with the next head of the writer queue because the value of *PHID* alternates between writers. PF-Q locks have $O(1)$ RMR complexity because neither readers nor writers spin on shared lock state.

In this section, we have shown that phase-fair locks can be implemented efficiently on common hardware platforms. We report on the results of an implementation-based performance study of the considered RW lock choices next.

¹⁴Since *ticket* holds the value of *rou*t before it was incremented, it is compared against *last* – 1.

5 Implementation and Evaluation

To compare the various synchronization options discussed above, we determined the HRT and SRT schedulability of randomly-generated task sets under both partitioned and global scheduling assuming the use of task-fair mutex, task-fair RW, and phase-fair RW group locks. The methodology followed in this study, and our results, are discussed below.

5.1 Test Platform

In order to capture the impact (or lack thereof) of RW synchronization as accurately as possible, we conducted our study under consideration of real-world system overheads as incurred in LITMUS^{RT}, UNC’s Linux-derived real-time OS (Brandenburg et al., 2007; Calandrino et al., 2006), on a Sun UltraSPARC T1 “Niagara” multicore platform.

The Niagara is a 64-bit machine containing eight cores on one chip clocked at 1.2 GHz. Each core supports four hardware threads, for a total of 32 logical processors. These hardware threads are real-time-friendly because each core distributes cycles in a round-robin manner—in the worst case, a hardware thread can utilize every fourth cycle. On-chip caches include a 16K (resp., 8K) four-way set associative L1 instruction (resp., data) cache per core, and a shared, unified 3 MB 12-way set associative L2 cache. Our test system is configured with 16 GB of off-chip main memory.

While the Niagara is clearly not an embedded systems processor, it is nonetheless an attractive platform for forward-looking real-time systems research. Its power-friendly combination of many simple and slow cores, predictable hardware multi-threading, and a small shared cache is likely indicative of future processor designs targeting computationally-demanding embedded systems.¹⁵ Thus, we believe any limitations exposed on the enterprise-class Niagara today to be of value as guidance to future embedded system designs. However, note that specific results, as with all implementation-based studies, directly apply only to the tested configuration.

5.2 Implemented Locks

After evaluating the bounds on worst-case blocking (see Appendix A), we chose to omit both reader and writer preference locks from our implementation-based performance study—due to extensive blocking, preference locks are not a competitive choice even if overheads are discounted. This is clearly demonstrated in Figure 9, which shows the average number of requests contributing to each task’s arrival blocking bound under G-EDF as a function of increasing contention. While jobs under task-fair mutex locks are blocked by at most $m = 32$ requests upon release, and even fewer requests under both task-fair and phase-fair RW locks, arrival blocking grows unboundedly under both preference lock types, and especially so under reader preference locks—at one request per resource per millisecond, the number of arrival blocking requests under reader preference locks is over seven times larger than under task-fair mutex locks. Thus, preference locks are not a viable choice for real-time systems that require *a priori* bounds on worst-case blocking delays.

¹⁵Similarly, 32-bit processors with L2 caches and speeds in excess of 100 MHz, once firmly associated with enterprise-class servers, are now routinely deployed in embedded systems.

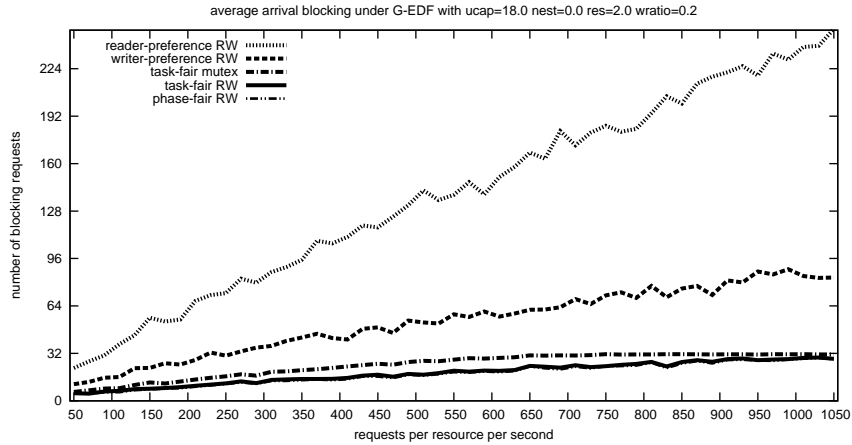


Figure 9: Average arrival blocking in terms of the number of blocking requests as a function of contention under each group lock choice (the curves for task-fair and phase-fair RW locks coincide). Note that under task-fair mutex, arrival blocking does not exceed $m = 32$, and that arrival blocking under both task-fair and phase-fair RW locks is less than under task-fair mutex locks. Due to starvation, arrival blocking under both reader and writer preference locks quickly exceeds m and never compares favorably to task-fair mutex locks. This graph corresponds to the *Single Read* and *Single Write* columns in Table 1 since arrival blocking is determined in large parts by the bound on worst-case blocking of a single request. (See Section 5.3 for a discussion of the experimental setup.)

Implementation efficiency. As discussed in Section 4, lock performance strongly depends on the underlying hardware characteristics and cannot in general be predicted without measuring actual implementations. Thus, in order to realize group locks efficiently, we implemented optimized versions of two local-spin mutex locks and two local-spin RW locks proposed by Mellor-Crummey and Scott (Mellor-Crummey and Scott, 1991a,b), as well as our phase-fair RW ticket and queue locks, in LITMUS^{RT}. As seen in Table 2, these locks are denoted MX-T, MX-Q, TF-T, TF-Q, PF-T, and PF-Q respectively.

We implemented each lock on both Intel’s x86 and Sun’s SPARC V9 architectures. Intel’s x86 architecture supports atomic-add and fetch-and-add directly via the `add` and `xadd` instructions. On Sun’s SPARC V9 architecture, a RISC-like design, only compare-and-swap is natively supported and hence both instructions are emulated. In the following, we focus on the implementations specific to the Niagara, but note that similar trends similar to those discussed below were also observed on x86-based systems.

Recall that lock algorithms are commonly classified by their RMR complexity (Anderson et al., 2003)—in theory, $O(1)$ locks, e.g., MX-Q, TF-Q, and PF-Q, should outperform $O(m)$ locks, e.g., MX-T, TF-T, and PF-T, since frequent cache invalidations cause memory bus contention. To test this assumption, we implemented a micro benchmark in which a configurable number of processors access a few shared variables—protected by one lock—repeatedly in a tight loop. The loop was configured via two parameters: *wratio*, which determined the fraction of updates, and *delay*, which determined the time between consecutive requests of one processor such that each processor spent approximately $\frac{1}{1+delay}$ of the total execution time in critical

Algorithm	Name	Complexity
Task-Fair Mutex Ticket Lock (Mellor-Crummey and Scott, 1991a)	MX-T	$O(m)$
Task-Fair Mutex Queue Lock (Mellor-Crummey and Scott, 1991a)	MX-Q	$O(1)$
Task-Fair RW Ticket Lock (Mellor-Crummey and Scott, 1991b)	TF-T	$O(m)$
Task-Fair RW Queue Lock (Mellor-Crummey and Scott, 1991b)	TF-Q	$O(1)$
Phase-Fair RW Ticket Lock (Listing 1)	PF-T	$O(m)$
Phase-Fair RW Queue Lock (Listings 3 and 4)	PF-Q	$O(1)$
Linux RW Lock (version 2.6.24)	LX	—

Table 2: Locking algorithms considered in this paper. The complexity metric is remote memory references on m cache-coherent processors. Linux’s RW lock implementation resembles a writer preference lock but does not offer strong progress guarantees; it serves only as a baseline for implementation performance.

sections. We varied the number of processors from $2, \dots, 32$ for each combination of $wratio \in \{0.01, 0.05, 0.1, 0.2, 0.35\}$ and $delay \in \{1, \dots, 10\}$ and recorded the time required for each processor to (concurrently) execute 200,000 loop iterations (after an initial warm-up phase) for each of the locks listed in Table 2.

Figures 10 and 11 show the measured results on our Niagara for $wratio = 0.1$ and $delay = 2$, *i.e.*, each processor spent about 33% of the execution time trying to access shared variables. Figure 10 shows comparisons of a ticket-based to a queue-based implementation of task-fair mutex, task-fair RW, and phase-fair RW group locks respectively. Figure 11 shows the same data, however the curves are grouped differently to allow a comparison of all considered queue locks in inset (a), and all considered ticket locks in inset (b). Further, each inset also shows Linux’s RW lock implementation as a baseline. Each graph shows the average critical section length (including synchronization overhead) normalized by the average critical section length if no locks are acquired, *e.g.*, a value of 1.5 means that 50% overhead was caused by synchronization. These graphs reveal several trends:

- none of the queue locks is preferable to its ticket-based counterpart, even under unrealistically high levels of contention (*e.g.*, see Figure 10);
- the gap between queue locks and ticket locks widens until about 20 processors contend for shared variables, and narrows afterwards as the reduced bus contention starts to pay off for queue locks with increasing contention;
- queue lock competitiveness decreases with increasing $delay$, *i.e.*, decreasing contention, since processors spend less time spinning (which queue locks are optimized for) and the lock entry and exit overhead is emphasized (which is high in queue locks, due to their complexity);
- mutex locks match (MX-Q) or outperform (MX-T) Linux’s RW lock if at most four processors contend since mutex locks have a simpler (and thus more efficient) implementation, but throughput quickly deteriorates due to the lack of reader parallelism (*e.g.*, see Figures 10(a) and 11);

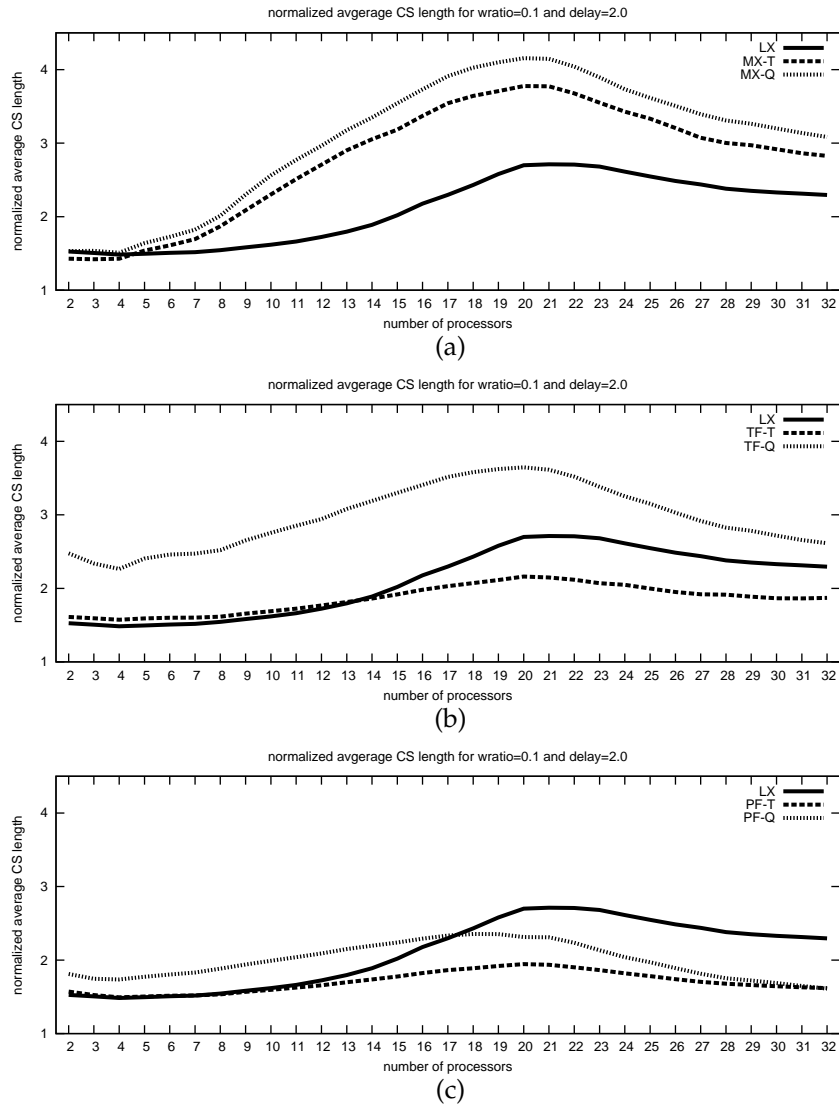


Figure 10: Graphs showing the average critical section lengths as a function of the number of contending processors on a Sun Niagara multicore processor. The measured average length is normalized with respect to the measured average length if synchronization is disabled. Comparison of (a) mutex locks; (b) task-fair RW locks; and (c) phase-fair RW locks. Each graph also shows the performance of Linux’s RW lock implementation (which does not offer strong progress guarantees) as a baseline. See Table 2 for a list of the considered lock types.

- of the tested RW locks, only the PF-T, TF-T, and PF-Q locks outperform Linux’s RW implementation, and only under high contention, *i.e.*, starting at around 10–17 processors (see insets (b) and (c) of Figure 10);
- PF-T is the only tested RW lock that performs at least as well as Linux’s RW implementation for all processor counts (*e.g.*, see Figures 10(c) and 11); and
- PF-Q is the only tested queue lock type that approaches ticket lock performance at high

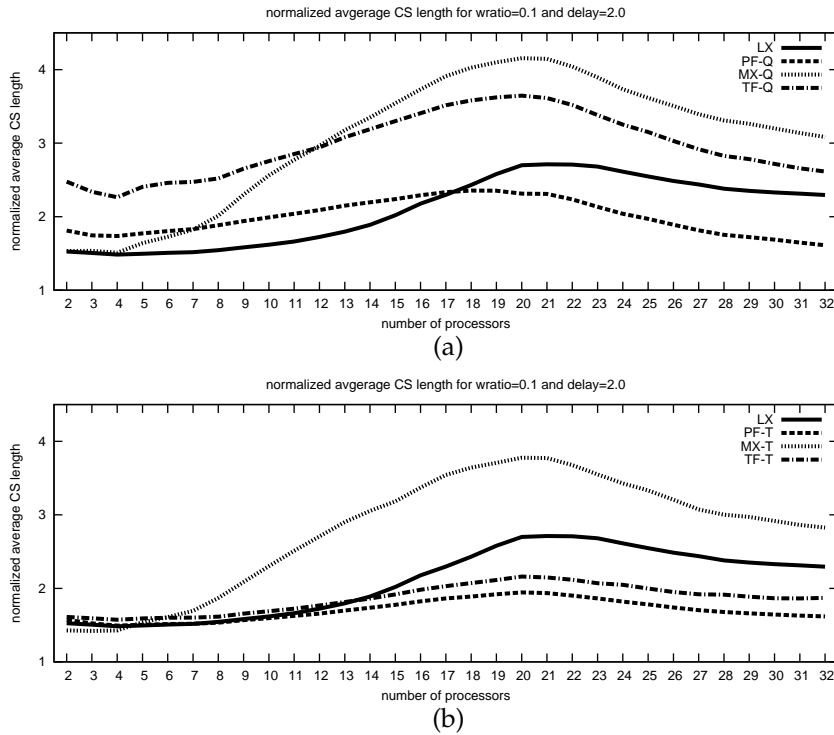


Figure 11: Graphs showing the average critical section lengths as a function of the number of contending processors on a Sun Niagara multicore processor. The measured average length is normalized with respect to the measured average length if synchronization is disabled. Comparison of (a) all considered queue locks; and (b) all considered ticket locks. Each graph also shows the performance of Linux’s RW lock implementation (which does not offer strong progress guarantees) as a baseline. See Table 2 for a list of the considered lock types.

processor counts (*e.g.*, see inset (c) of Figure 10).

We attribute the good performance of both phase-fair locks to their simplicity, and especially so in the case of the PF-Q lock, which has significantly simpler queue handling requirements than the TF-Q lock.

The observed competitiveness of mutex locks if at most four processors contend likely benefits from a peculiarity of our test platform, which realizes the first four logical processors as hardware threads of the same physical core—execution is thus implicitly serialized and the L1 cache is shared among all contending processors.

Conclusions. It should be noted that this micro benchmark is limited in scope and that the observed trends do not necessarily extend to other platforms (*e.g.*, queue locks may be more competitive on platforms with non-uniform memory access costs) or other workloads (*e.g.*, performance differences may be marginal if critical sections are very long or if the majority of the critical sections are writes).

For the purpose of this paper, however, our results (*e.g.*, Figure 10) show that—on our test platform, and for short critical sections—none of the queue locks are preferable to their ticket-based counter-parts under even unrealistically high levels of contention, and incur significantly

higher overheads if only few processors contend. This is because queue locks require more atomic compare-and-swap instructions than their ticket counterparts, and atomic operations execute very slowly on our test platform (and most other architectures in widespread use) due to architectural atomicity and cache-consistency issues that are beyond the scope of this paper. Thus, while ticket-based locks are of asymptotically higher RMR complexity, they are preferable on real hardware due to lower constant factors.

To summarize, the positive effect of $O(1)$ RMR complexity (*i.e.*, reduced bus traffic) only starts to outweigh the associated increased entry and exit overhead at unrealistic contention levels, *i.e.*, accesses to shared resources must comprise more than 33% of each processor’s load in order to break even (*e.g.*, see the curves for PF-T and PF-Q in Figure 10(c), where $delay = 2$), which we consider rather unlikely to occur in well-designed real-time systems.

Hence, we focus exclusively on the MX-T, TF-T, and PF-T locks throughout the remainder of this paper, as their queue-based counter-parts are subject to identical blocking analysis (respectively) but incur higher overheads, and thus cannot possibly yield higher schedulability. Having selected the algorithms to evaluate, we describe our experimental setup in detail next.

5.3 Task Set Generation

When generating random task sets for conducting schedulability comparisons, task parameters were selected—similar to the approach previously used in (Brandenburg and Anderson, 2008; Brandenburg et al., 2008a,b)—as follows. In the main study, task utilizations were distributed uniformly over $[0.1, 0.4]$, periods were chosen from $[10ms, 100ms]$, and task were assumed to have implicit deadlines, *i.e.*, $p(T_i) = d(T_i)$. We selected these parameter ranges because they correspond to the “medium weight distribution” previously considered in (Brandenburg et al., 2008a). We consider the impact of allowing shorter and longer periods, higher task utilizations, and constrained deadlines in Section 5.8. Task execution costs were calculated based on utilizations and periods. Periods were defined to be integral, but execution costs may be non-integral. Task sets were obtained by generating tasks until a *utilization cap* ($ucap$) was reached.

5.4 Resource Sharing

Resources and requests were generated based on four parameters. The total number of generated tasks N and the *average number of resources per task* (res) was used to determine the total number of resources $R = N \cdot res$. Based on R and the *average number of requests per resource per second* (*contention*), the *total request density* $Q = R \cdot contention$ was computed. Note that request density is a normalized measure similar to task utilization and not necessarily integral. Q was further split based on the *ratio of write requests* ($wratio$) into *write density*, $Q_w = Q \cdot wratio$, and *read density*, $Q_r = Q \cdot (1 - wratio)$.

In the next step, we generated read (write) requests and randomly assigned them to tasks until the total *request density* of the generated read (write) requests equaled Q_r (Q_w). Request density is the normalized rate at which a request is issued.¹⁶ Based on the *nesting probability*

¹⁶We generated *sporadic request patterns*, as described in detail in Appendix A. If every k^{th} job of T_i issues a request X for a particular resource, then X has a density of $\frac{1000ms}{p(T_i) \cdot k}$. The factor of 1000ms is due to the use of one second as the normalization interval. We chose $k = 1$ unless that would have exceeded Q_r (Q_w). Larger values of

Overhead	Worst-Case	Average-Case
MX-T: read/write request	0.130 μ s	0.129 μ s
TF-T: read request	0.160 μ s	0.157 μ s
TF-T: write request	0.154 μ s	0.153 μ s
PF-T: read request	0.144 μ s	0.142 μ s
PF-T: write request	0.180 μ s	0.178 μ s
leaving non-preemptive section	2.137 μ s	1.570 μ s

Table 3: Worst-case and average-case synchronization overheads. Based on the methodology explained in detail in (Brandenburg et al., 2008a), the results were obtained by computing the maximum (resp. average) cost of 1,000,000 requests after discarding the top 1% to remove samples that were disturbed by interrupts and other outliers.

(*nest*), each request contained d levels of nested requests with a probability of $nest^d$. Resource groups as mandated by the FMLP were computed during request generation. Due to the randomized assignment of requests, jobs of some tasks may not request resources at all, and jobs of some tasks may both read and write the same resource. However, we ensured that each resource is requested by at least one writer and one reader that are not identical. The duration of requests was distributed uniformly in $[1.0\mu s, 15.0\mu s]$. This range was chosen to correspond to request durations considered in prior studies on mutex synchronization (Brandenburg and Anderson, 2008; Brandenburg et al., 2008b), which in turn were based on durations observed in actual systems (Brandenburg and Anderson, 2007). The effect of allowing longer durations is discussed in Section 5.8.

5.5 Overheads

We used task-centric interrupt accounting to accommodate job release overhead (Brandenburg et al., 2009). Other system and synchronization overheads were accounted for by inflating worst-case execution costs and the durations of outermost requests using standard techniques (Liu, 2000). Most relevant system overheads (*e.g.*, scheduling overhead, context-switch overhead, *etc.*) were already known from a recent study (Brandenburg et al., 2008a) and did not have to be re-determined. Only synchronization-related overheads had to be obtained and are given in Table 3. When determining HRT (SRT) schedulability, we assumed worst-case (average-case) overheads, as previously done in (Brandenburg et al., 2008a).

Since research on timing analysis has not matured to the point of being able to analyze complex interactions between tasks due to atomic operations, bus locking, and bus and cache contention, overheads must be determined experimentally. As such, one should *not* consider these overhead values to accurately represent “true” worst case values. Since the focus of this paper is not worst-case timing analysis, and since these overhead figures are only used as an approximate notion of implementation efficiency, we consider this to be an acceptable tradeoff.

k are common for low *wratios*.

5.6 Schedulability Tests

After a task system was generated and all overheads accounted for, its schedulability assuming MX-T, TF-T, and PF-T group locks was tested as follows. Per-task bounds on worst-case blocking were computed as explained in Appendix A, and each task’s worst-case execution cost was inflated to account for the corresponding utilization loss due to spinning.

Under G-EDF, a task system was deemed SRT schedulable if the total utilization *after inflation* did not exceed $m = 32$ (Devi and Anderson, 2008). Determining whether a task system is HRT-schedulable under G-EDF is more involved. There are now five major sufficient (but not necessary) HRT schedulability tests for G-EDF (Baker, 2003; Baruah, 2007; Bertogna et al., 2005, 2009; Goossens et al., 2003). Interestingly, for each of these tests, there exist task sets that are deemed schedulable by it but not the others (Baruah, 2007; Bertogna et al., 2009). Thus, a task system was deemed HRT schedulable under G-EDF if it passed at least one of these five tests.

Under P-EDF, a task system was deemed schedulable if it could be partitioned using the *worst-fit decreasing heuristic* and the utilization after inflation did not exceed one on any processor (Liu and Layland, 1973). Note that partitioning must precede the blocking-term calculation. Note also that HRT and SRT schedulability under P-EDF is the same except for the use of worst-case versus average-case overheads.

5.7 Study

In our study, we assessed SRT/HRT schedulability under both G-EDF and P-EDF while varying each of the following five task set generation parameters (as described in Sections 5.3 and 5.4 above):

- the total utilization cap, $ucap \in [1.0, 32.0]$,
- the average number of requests per resource per second, $contention \in [50, 1050]$,
- the ratio of write requests, $wratio \in [0.01, 0.5]$,
- the probability of nested requests, $nest \in [0.0, 0.5]$, and
- the average number of resources per task, $res \in [0.1, 7.0]$.

While varying one parameter (*i.e.*, dimension), we chose constant values for all other parameters in order to obtain two-dimensional graphs. Thus, each of the above parameters was varied over its stated range for *all* possible choices of the other parameters arising from

- $ucap \in \{9.0, 12.0, 15.0\}$ for HRT and $ucap \in \{15.0, 18.0, 21.0\}$ for SRT,
- $contention \in \{100, 250, 400\}$,
- $wratio \in \{0.05, 0.1, 0.2, 0.35\}$,
- $nest \in \{0.0, 0.05, 0.2, 0.35\}$, and
- $res \in \{0.5, 2.0, 3.5\}$,

under both SRT and HRT and both G-EDF and P-EDF scheduling.

For example, consider the scenario depicted in Figure 12(a). To obtain said graph, task sets were randomly generated as described in Sections 5.3 and 5.4 assuming $contention = 400$, $wratio = 0.2$, $nest = 0.0$, and $res = 3.5$ while sampling different values of $ucap$ in the range from 1.0 to 32.0.

Sampling points were chosen such that the sampling density is high in areas where curves change rapidly. For each sampling point, we generated (and tested for schedulability) 50 task sets, for a total of over 1,600,000 tested task sets.

Trends. It is clearly not feasible to present all 2,592 resulting graphs. However, the results show clear trends. We begin by making some general observations concerning these trends. Below, we consider a few specific graphs that support our observations.

In the majority of the tested scenarios, PF-T locks were the best-performing algorithm. MX-T locks were preferable in some of the tested scenarios involving high write ratios (as discussed below).

Due to the wide range of parameter values considered, some parameter combinations did not exhibit discernible trends since they were either “too easy” (low $contention$, res , $ucap$) or “too hard” (high $ucap$, $contention$, $wratio$, $nest$). In these scenarios, either (almost) all or none of the task sets were schedulable regardless of the group lock type, or synchronization performance did not significantly affect schedulability—this occurred mostly when varying $ucap$ under G-EDF, since G-EDF suffers from limited HRT schedulability even when tasks are independent (Brandenburg et al., 2008a). Where RW synchronization was clearly preferable to mutual exclusion, TF-T locks *never* outperformed PF-T locks.

Generally speaking, PF-T locks were usually more resilient to increases in $contention$, res , $ucap$, $nest$, and $wratio$, *i.e.*, they exhibited higher schedulability than the other two choices under increasingly adverse conditions. Hence, it is more illuminating to consider the two exceptions: (i) under which conditions are MX-T locks preferable to PF-T locks (and why), and (ii) under which conditions do TF-T locks perform as well as PF-T locks? Regarding (i), all cases exhibit a combination of high request density, deep and frequent nesting, and many writers ($wratio \geq 0.25$). Hence, writer blocking, which is not improved by RW locks, becomes the dominating performance factor. This explains why PF-T (and TF-T) locks do not perform better than MX-T locks in some cases, but why do PF-T locks sometime perform *worse*? The reason is that PF-T locks incur higher arrival blocking in the presence of frequent writes (discussed below), whereas TF-T locks degrade only to mutex-like behavior.

The answer to (ii) reinforces the intuition that task-fair locks are very sensitive to the number of concurrent writers: *all* scenarios in which TF-T locks perform as well as PF-T locks (and in which RW synchronization is preferable to mutual exclusion) exhibit a very low write density ($wratio = 0.05$). However, the inverse is not true: there are scenarios in which PF-T locks clearly perform better than TF-T locks (in terms of schedulability) where $wratio = 0.05$.

Example graphs. Figures 12–20 display graphs corresponding to 15 selected scenarios that illustrate the above trends. Each of the Figures 12–16 shows two graphs with superior PF-T performance for each of the five parameters that we varied. In order to show a wide variety of scenarios, we chose to exhibit one HRT schedulability result under P-EDF (inset (a) in Fig-

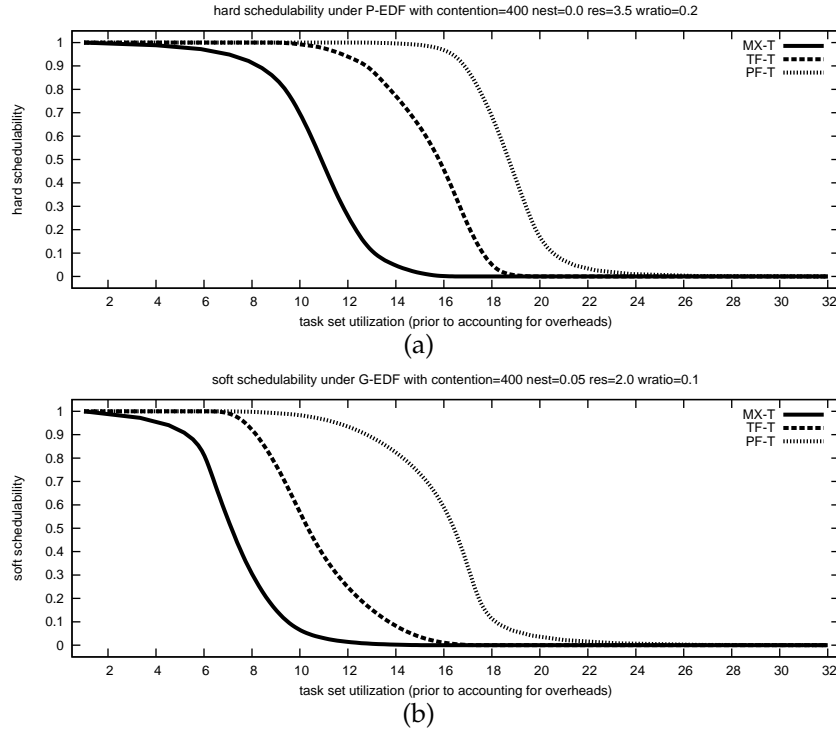


Figure 12: Schedulability as a function of task system utilization ($ucap$). **(a)** Hard schedulability (zero tardiness under worst-case overheads) under P-EDF; and **(b)** soft schedulability (bounded tardiness under average-case overheads) under G-EDF. The y -axis gives the fraction of successfully-scheduled task sets. The considered scenario is indicated above each graph.

ures 12–16) and one SRT schedulability result under G-EDF (inset (b) in Figures 12–16) for each parameter. Further, Figures 17–20 illustrate scenarios with degraded PF-T performance. For presentation purposes, the selected example graphs have been plotted using Gnuplot’s Bezier interpolation.

Figure 12 shows schedulability as a function of $ucap$. In both insets, PF-T locks yield significantly better schedulability: in inset (a), under P-EDF with 20% writes and no resource nesting, performance starts to degrade under MX-T and TF-T locks at $ucap \approx 8$ and $ucap \approx 10$ (resp.), whereas PF-T locks can sustain high schedulability until $ucap \approx 16$ —a 100% improvement over MX-T locks. Similarly, in inset (b), under G-EDF with 10% writes and some nesting ($nest = 0.05$), PF-T locks achieve 90% schedulability until $ucap \approx 14$, whereas TF-T quickly starts to degrade already at $ucap \approx 8$.

Figure 13 shows schedulability as a function of $contention$. For both P-EDF and G-EDF, PF-T locks can sustain almost twice as much contention as MX-T locks before performance degrades. This highlights the significance of the PF-T lock’s $O(1)$ bound on read blocking under high contention. While TF-T locks offer some advantage over MX-T locks, they can support only about 60% of the contention that PF-T locks support. Thus, under conditions favorable to RW locks (no nesting, moderate $wratio$), PF-T locks exhibit better scalability. Note that scenarios exist in which PF-T locks do not significantly decrease in performance until 1050 requests per second—TF-T locks did not support such high levels of $contention$.

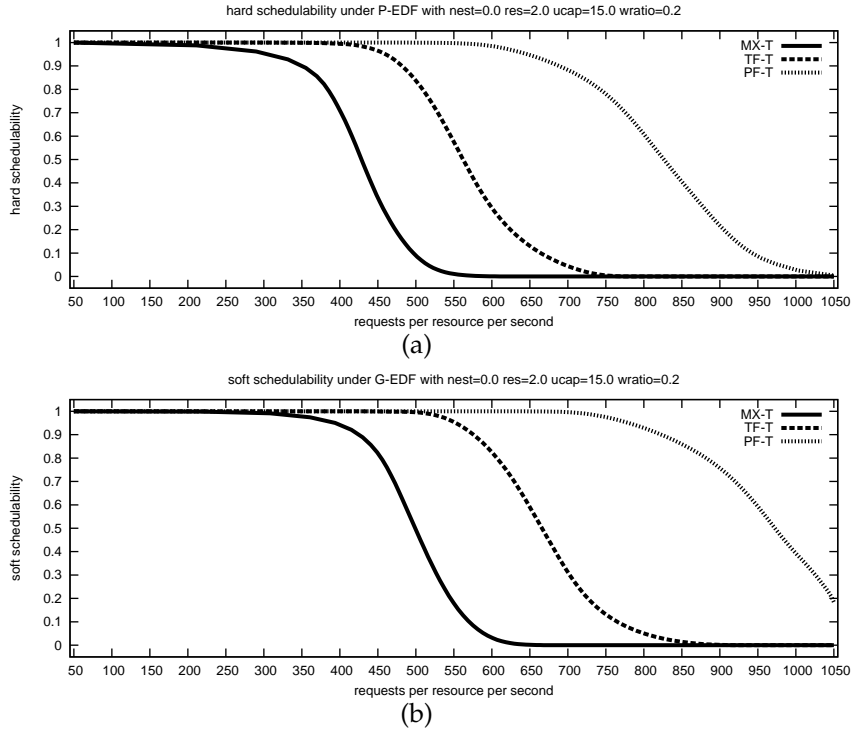


Figure 13: Schedulability as a function of the average requests per resource per second (*contention*). **(a)** Hard schedulability (zero tardiness under worst-case overheads) under P-EDF; and **(b)** soft schedulability (bounded tardiness under average-case overheads) under G-EDF. The y -axis gives the fraction of successfully-scheduled task sets. The considered scenario is indicated above each graph.

Figure 14 shows schedulability as a function of $wratio$ under conditions that are favorable to RW locks. Under both P-EDF and G-EDF, schedulability is virtually zero from the outset with MX-T locks, and degrades quickly under TF-T locks. In inset (b), PF-T locks can sustain high schedulability until $wratio \approx 0.3$, whereas TF-T locks start to degrade around $wratio = 0.15$. This highlights that a small number of writers affects TF-T locks much more severely than PF-T locks.

Figure 15 shows schedulability as a function of $nest$. In both cases, PF-T locks perform significantly better than either MX-T and TF-T locks as resource groups become fewer in number and larger, even under high *contention* (G-EDF case) and moderate $wratio$ (P-EDF case). Especially in the G-EDF case, TF-T performance resembles MX-T performance much more closely than PF-T performance—in spite of an RW-friendly $wratio$ value of 0.05.

Finally, Figure 16 shows schedulability as a function of res . Once again, in inset (a), PF-T locks can sustain significantly higher schedulability whereas TF-T locks perform only little better than MX-T locks. In inset (b), under G-EDF, PF-T locks can sustain about three times as many resources per task than MX-T locks, and more than 1.5 times as many resources per task than TF-T.

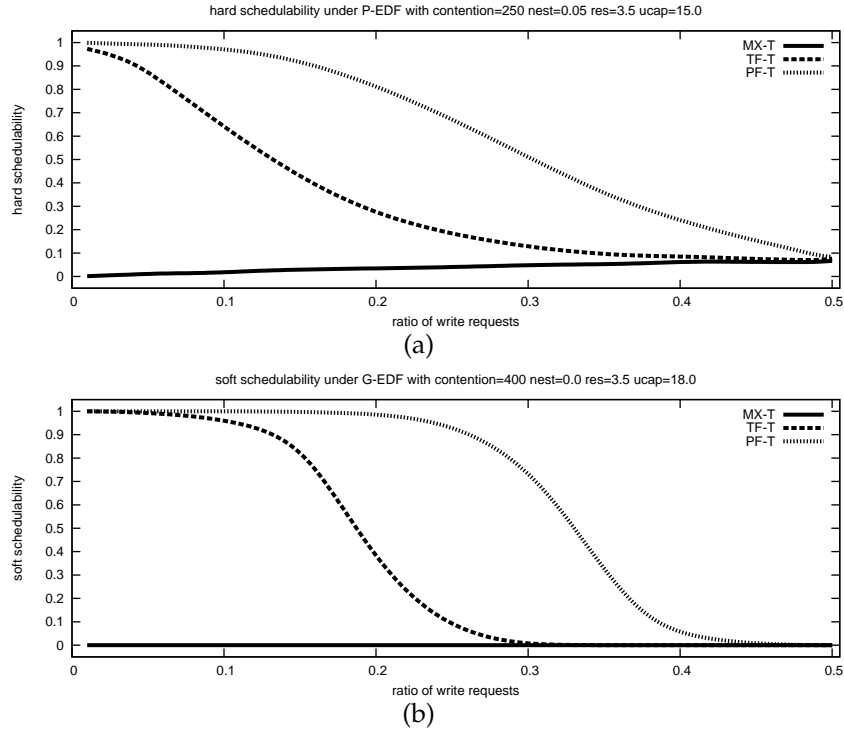


Figure 14: Schedulability as a function of the ratio of write requests ($wratio$). **(a)** Hard schedulability (zero tardiness under worst-case overheads) under P-EDF; and **(b)** soft schedulability (bounded tardiness under average-case overheads) under G-EDF. The y -axis gives the fraction of successfully-scheduled task sets. The considered scenario is indicated above each graph.

Limitations. Figures 12–16 show that schedulability can be improved significantly by employing phase-fair locks, an observation that is well supported by a large majority of the 2,592 considered scenarios. However, phase-fair locks can also perform worse than MX-T locks under certain conditions. This is illustrated by Figures 17–20, which exhibit examples of inferior PF-T performance under G-EDF.

Figure 17 shows inferior SRT schedulability under PF-T locks when varying $ucap$ (inset (a)) and $contention$ (inset (b)). Similar trends can be found in Figure 18, which shows SRT schedulability as function of res (inset (a)) and $nest$ (inset (b)). Note that all four scenarios are quite similar:

- very high write density ($wratio = 0.35$),
- very high degree of nesting in Figure 17 and Figure 18(a) ($nest = 0.35$), and thus
- virtually no benefit from employing RW locks (MX-T and TF-T curves overlap).

In essence, the high ratio of write requests makes RW synchronization undesirable in these scenarios. This is confirmed by the graph depicted in Figure 19(a), which shows SRT schedulability as a function of $wratio$ under deep nesting $nest = 0.35$. While first performing better than TF-T locks in the range from $wratio \approx 0.02$ to $wratio \approx 0.2$, performance under PF-T locks quickly becomes worse than MX-T performance as $wratio$ starts to exceed 25%. Note that this reveals

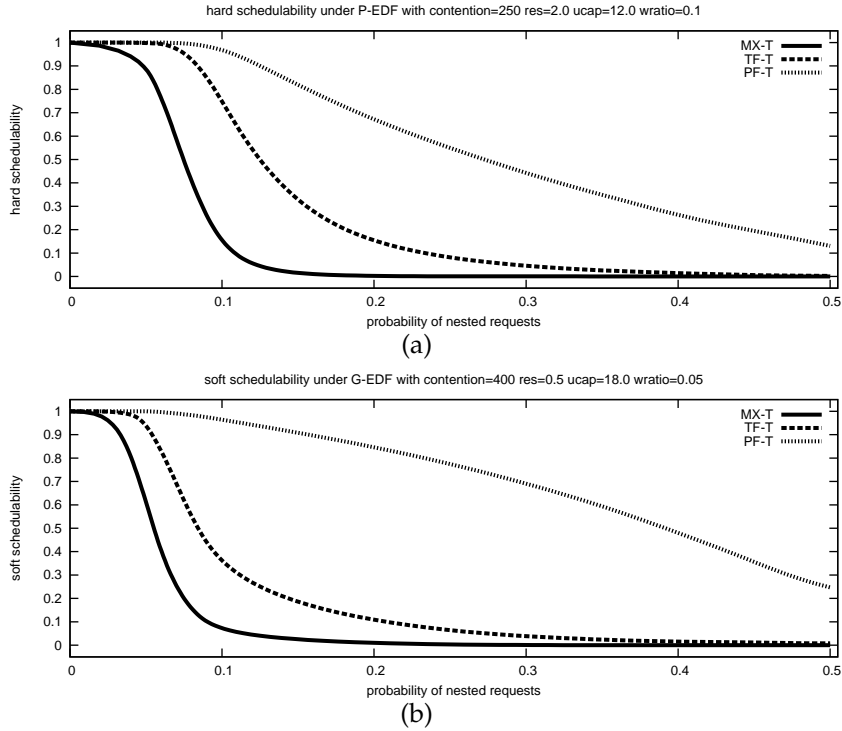


Figure 15: Schedulability as a function of the probability of nested requests (*nest*). **(a)** Hard schedulability (zero tardiness under worst-case overheads) under P-EDF; and **(b)** soft schedulability (bounded tardiness under average-case overheads) under G-EDF. The *y*-axis gives the fraction of successfully-scheduled task sets. The considered scenario is indicated above each graph.

a fundamental difference between task-fair and phase-fair RW locks—the former degrade to mutex performance, whereas the latter can perform worse.

This is further supported by Figure 19(b), which depicts the same scenario as Figure 19(a), but shows the average per-task utilization inflation due to blocking (both direct and arrival) instead of schedulability. Note that the points where the per-task inflation curves cross in Figure 19(b) correspond to the points where schedulability curves cross in Figure 19(a), and also note how the rate of change of the TF-T inflation curve decreases to match the MX-T inflation curve, whereas the PF-T inflation curve grows linearly with increasing *wratio*.

Figure 20 further explores this scenario: inset (a) shows average per-task direct blocking (in milliseconds); inset (b) shows average per-task arrival blocking (also in milliseconds). Note that direct blocking is much lower under PF-T locks, even for *wratio* > 0.25, but that arrival blocking is twice as large under PF-T locks than under MX-T locks. Further, arrival blocking exhibits only little growth, but direct blocking reveals a clear linear trend.

Thus, we can characterize PF-T performance in such scenarios as follows: due to a high degree of nesting, outermost requests have long durations, which exacerbates the impact of the factor of two in PF-T’s arrival blocking bound (recall the *Single Writer* column in Table 1). However, the increased arrival blocking is compensated by PF-T’s much lower average direct blocking if *wratio* < 0.25.

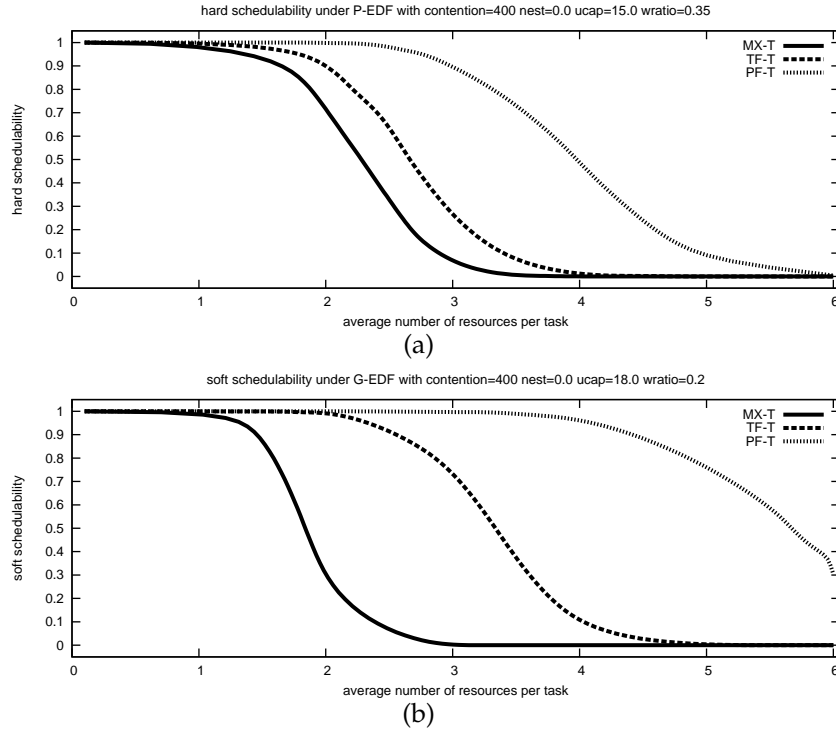


Figure 16: Schedulability as a function of the average number of resources per task (res). **(a)** Hard schedulability (zero tardiness under worst-case overheads) under P-EDF; and **(b)** soft schedulability (bounded tardiness under average-case overheads) under G-EDF. The y -axis gives the fraction of successfully-scheduled task sets. The considered scenario is indicated above each graph.

Note that such performance degradation only occurs for combinations of both deep nesting and frequent writes: in Figure 16(a), PF-T performance is superior even though $wratio = 0.35$, and, similarly, PF-T performance only starts to degrade for $nest > 0.15$ in Figure 18(b). Thus, PF-T locks offer significantly superior performance if the key assumption underlying the RW-FMLP are met (infrequent nesting, infrequent writes), but can degrade in pathological cases in which said assumptions are violated.

5.8 Varying Periods, Deadlines, Weights, and Request Lengths

We conducted additional experiments based on the scenarios displayed in Figures 12–16 in which the task set generation procedure was altered to assess the impact of allowing shorter or longer periods, constrained deadlines (*i.e.*, $d(i) \leq p(i)$), higher task utilizations, and longer critical sections. Some of the resulting graphs, which correspond to Figures 13(a) and 13(b) respectively, are depicted in Figures 21 and 22.

Figure 21(a) shows the impact of choosing periods uniformly from $[50ms, 250ms]$. Perhaps counter-intuitively, allowing longer periods causes bounds on blocking to become more pessimistic, and thus schedulability under each scheme is reduced. This is because lengthening periods causes a decrease in average per-request density, which in turn causes the average

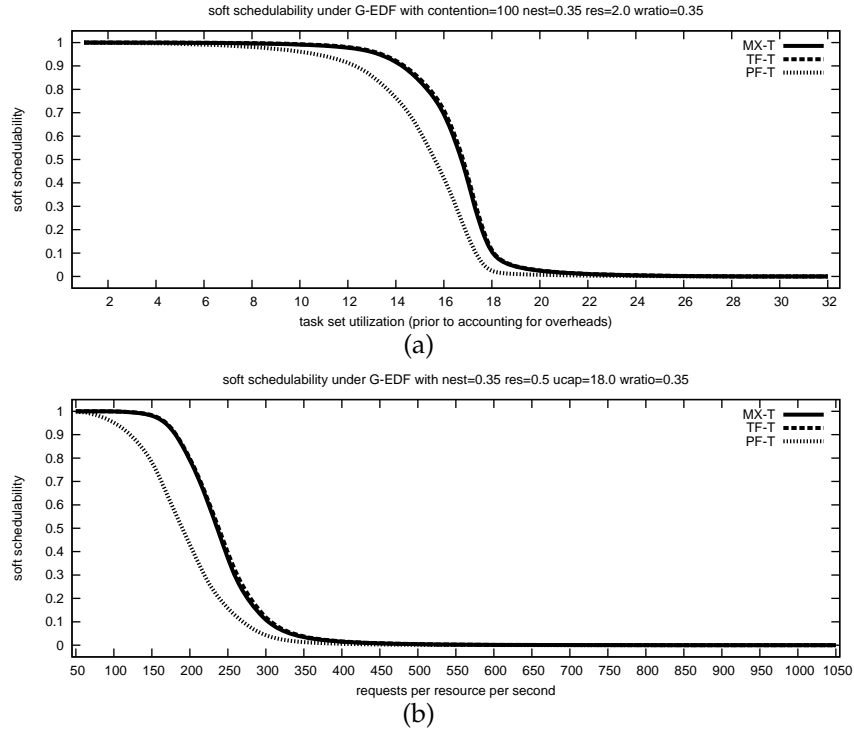


Figure 17: Soft schedulability under G-EDF as a function of **(a)** task system utilization ($ucap$); and **(b)** the average number of requests per resource per second ($contention$). The y -axis gives the fraction of successfully-scheduled task sets. The considered scenario is indicated above each graph.

number of requests per job to increase.

Figure 21(b) shows the impact of choosing periods uniformly from $[3ms, 33ms]$. Here, the trend is reversed—the average number of requests per job decreases, and thus schedulability is improved under each lock type. Of course, if periods are shortened to the point that there is insufficient slack to account for scheduling and synchronization overheads, then schedulability will be poor under any scheme.

An example of the effects of constraining each task’s relative deadline is shown in Figure 21(c). Compared to Figure 13(a), schedulability is slightly reduced under each lock type as deadline violations become more likely due to the stricter constraints.

Figure 22(a), which corresponds to Figure 13(b), shows the impact of allowing per-task utilizations up to 0.95 under G-EDF. Schedulability is drastically reduced under each lock type. This is because system overheads and blocking can, for some task sets, cause a task’s utilization to be inflated by more than 0.05. This is especially the case with tasks that have short periods. Thus, individual tasks may become over-utilized, which in turn limits schedulability. Since the utilization loss grows with average contention, far fewer requests can be supported in total.

Similar reasoning applies to Figure 22(b), which depicts schedulability assuming critical sections lengths in the range from $10\mu s$ to $50\mu s$. Increasing the duration of critical sections directly causes the bounds on worst-case blocking to become more pessimistic. Hence, the level of contention that can be sustained is reduced under each lock type, which highlights the

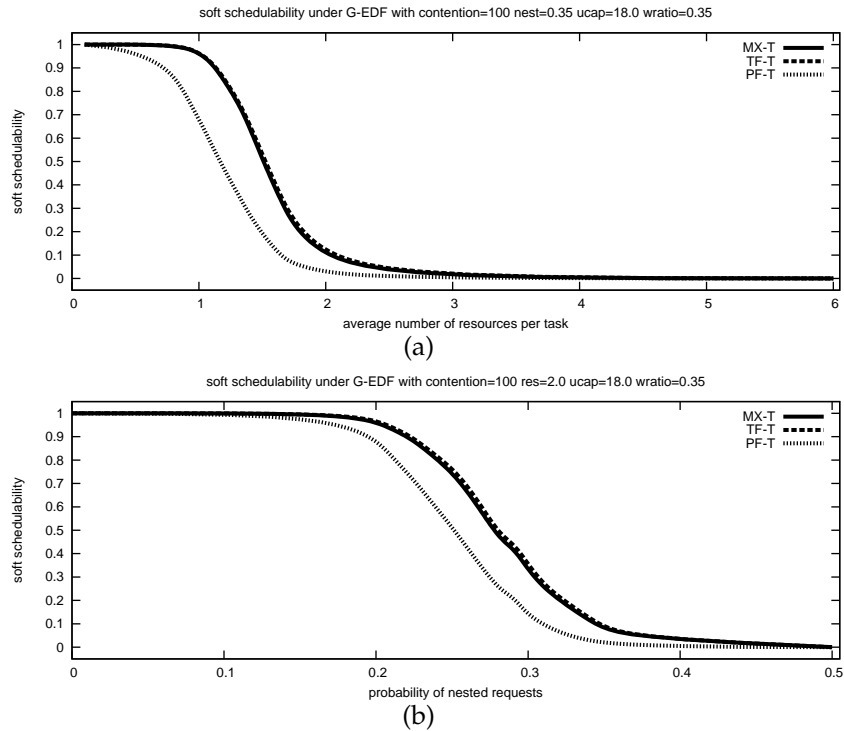


Figure 18: Soft schedulability under G-EDF as a function of **(a)** the average number of resources per task (res); and **(b)** the probability of nested requests ($nest$). The y -axis gives the fraction of successfully-scheduled task sets. The considered scenario is indicated above each graph.

benefit of keeping critical sections short.

Note that the relative performance of MX-T, TF-T, and PF-T locks remains generally unchanged in Figures 21 and 22 as all lock types are affected similarly by the changed task set generation procedure.

5.9 Portability Concerns

The RW-FMLP can easily be implemented as a library on top of POSIX-compliant operating systems such as Linux, VxWorks, or QNX.

For efficiency reasons, group locks should be implemented in userspace (*i.e.*, without resorting to system calls) using one of the phase-fair lock implementations presented in Section 4. If resources are to be shared across address space boundaries, then group locks must be allocated in shared memory (*e.g.*, as part of a memory-mapped file). Assigning resources to resource groups can be done either offline by the system designer, or automatically during an initialization phase before real-time execution starts (this requires all resource handles to be opened during initialization).

The RW-FMLP crucially depends on non-preemptive execution to bound the length of priority inversion (see Appendix A). Unfortunately, the POSIX standard does not define a standard API for jobs to declare non-preemptive sections. However, such functionality can be trivially

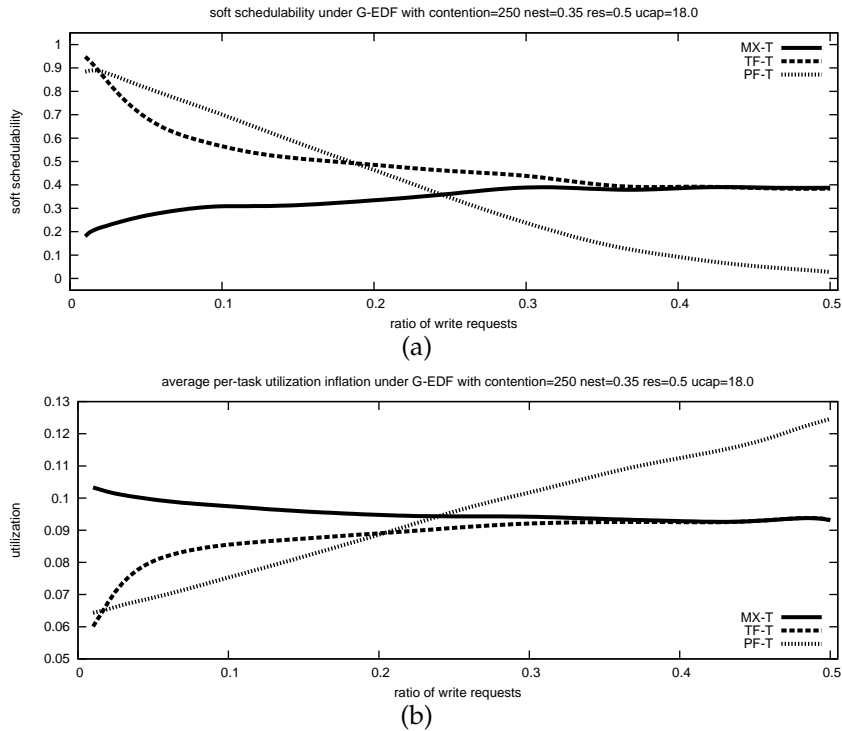


Figure 19: Two views of one scenario under G-EDF: **(a)** soft schedulability and **(b)** average per-task utilization inflation due to blocking as functions of the the ratio of write requests (*wratio*). Note that the two graphs are symmetrical—as the average per-task inflation increases, the average schedulability decreases. Two additional views of this scenario are shown in Figure 20.

emulated using static-priority¹⁷ scheduling: in order to avoid being preempted, a job can simply raise its priority to the highest real-time priority. Analogously, a job can re-enable preemptions at the end of a non-preemptive section by restoring its original priority.

With this straightforward emulation of non-preemptive sections, the RW-FMLP can be implemented on top of any OS that supports static-priority scheduling. However, this method is likely to require a system call each time that a job’s priority is altered. Given that system calls can incur significant overhead, a more efficient implementation is desirable. Therefore, the implementation in LITMUS^{RT} follows a different approach (Brandenburg et al., 2007, 2008b). To avoid the need for system calls in the common case (no preemption required) entirely, jobs signal non-preemptive sections to the scheduler by setting a flag in a control word that is shared between the kernel and userspace when entering a non-preemptive section, and clearing said flag on exit. Similarly, the scheduler sets a second flag in the shared control word when a required preemption was delayed and the currently-executing job should yield. Hence, under the RW-FMLP in LITMUS^{RT}, readers and writers incur no system call overhead when issuing a short resource request in the common case, and must issue only a single system call (to yield) in the rare event of a delayed preemption.

¹⁷Denoted `SCHED_FIFO` by the POSIX 1003.1b standard.

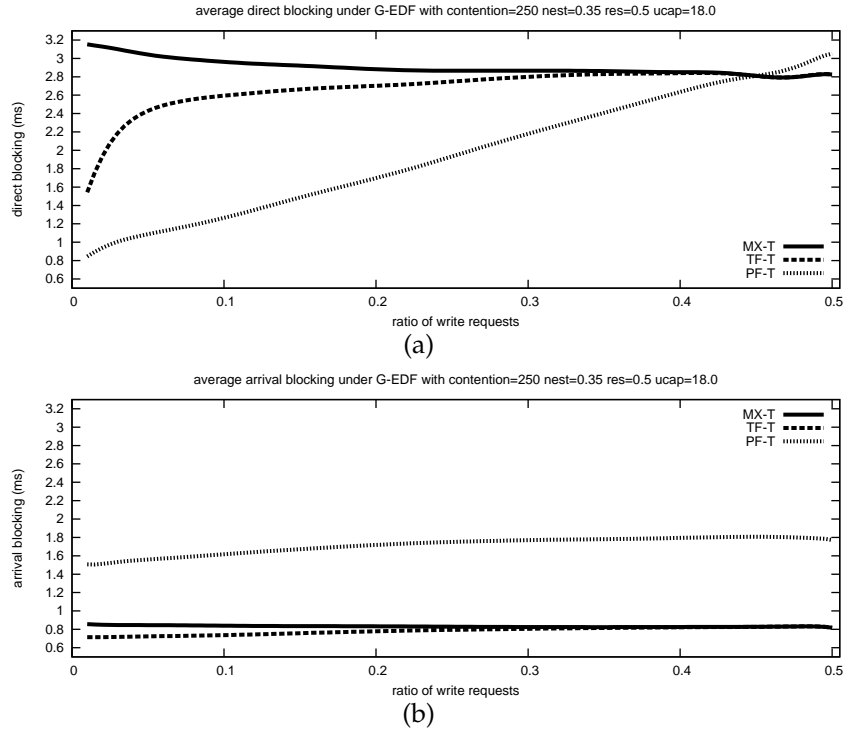


Figure 20: Two additional views of the scenario depicted in Figure 19: **(a)** average per-task direct blocking (in milliseconds) and **(b)** average per-task arrival blocking (in milliseconds) as functions of the the ratio of write requests ($wratio$). Note that arrival blocking is twice as high under phase-fair locks than under either task-fair mutex or RW locks, which is consistent with the constant factor of two in the *Single Write* column in Table 1. Further note that the difference between task-fair RW and task-fair mutex locks disappears as the write ratio increases, and rapidly so in the range from 0.01–0.05 in inset (a)—in the presence of frequent writes, task-fair RW locks degrade to mutex-like performance. Finally, note that phase-fair RW locks offer much reduced direct blocking compared to task-fair locks, unless the write ratio is high.

6 Conclusion

We have presented the first analysis of reader-writer locking in shared-memory multiprocessor real-time systems and have proposed phase-fair locks, a new RW lock design with asymptotically lower worst-case read blocking, and presented three phase-fair RW locks that can be implemented efficiently on common hardware platforms.

Our experiments revealed that (in terms of schedulability) phase-fair locks are frequently a superior choice if no more than 25%–35% of the requests are writes—oftentimes by a significant margin. In comparison to task-fair RW locks, phase-fair RW locks usually offer (much) better schedulability if there is benefit to employing RW locks at all, but can perform worse in pathological cases. To summarize:

- preference locks are not a viable choice for multiprocessor real-time systems,
- task-fair RW locks can degrade quickly to mutex-like performance, and

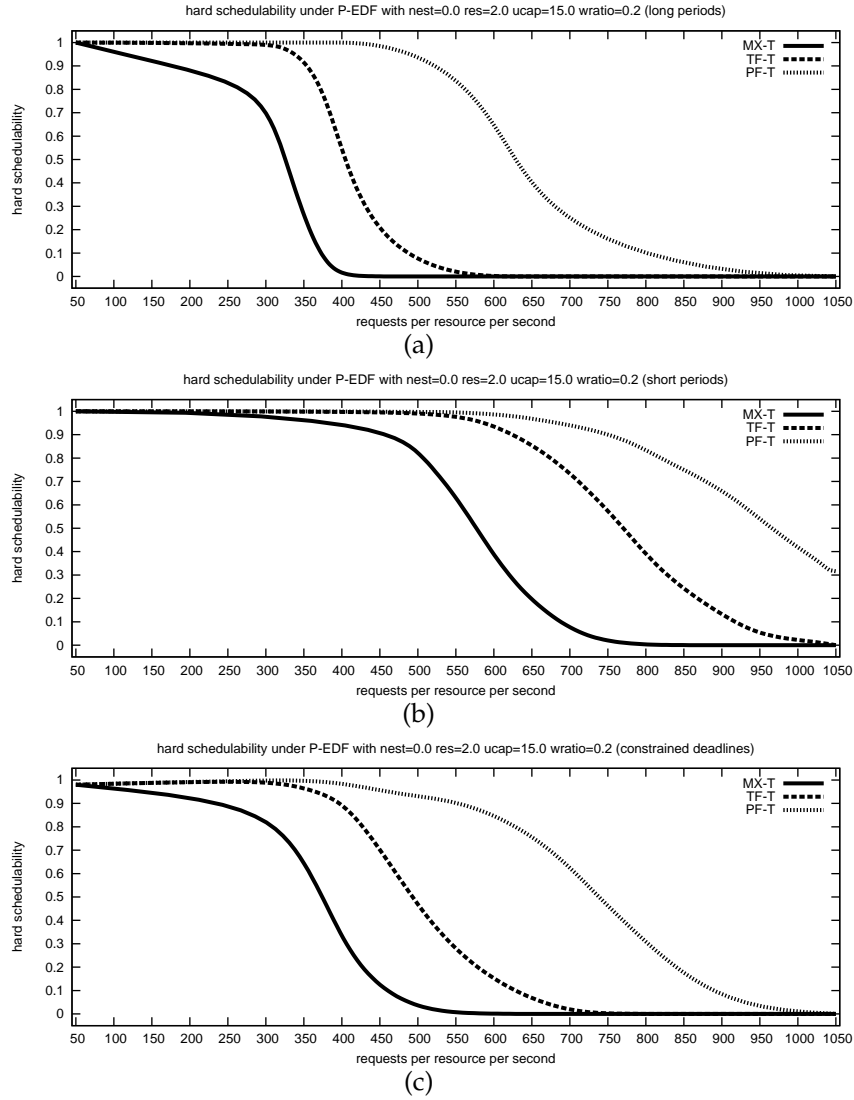


Figure 21: Three variations of the scenario depicted in Figure 13(a), wherein the task set generation procedure was altered by **(a)** uniformly choosing task periods from $[50ms, 250ms]$ (instead of $[10ms, 100ms]$), **(b)** uniformly choosing task periods from $[3ms, 33ms]$ (instead of $[10ms, 100ms]$), and **(c)** constraining relative deadlines such that per-task *static slack* was reduced by 25% to 75%, *i.e.*, for each task T_i , a parameter x was chosen uniformly from $[0.25, 0.75]$ to determine the relative deadline according to the formula $d(i) \triangleq e(i) + x \cdot (p(i) - e(i))$.

- phase-fair locks are usually the best choice, but can degrade to performance worse than that of task-fair mutex locks if both of the assumptions underlying phase-fairness are violated.

In future work, we plan to extend the RW-FMLP to support long resources by incorporating RW semaphores. Intuitively, phase-fairness should also reduce reader delay when blocking is realized by suspending; however, deriving *competitive* bounds on worst-case blocking is a challenge since the number of contending jobs is no longer bound by $m - 1$. Further, we would

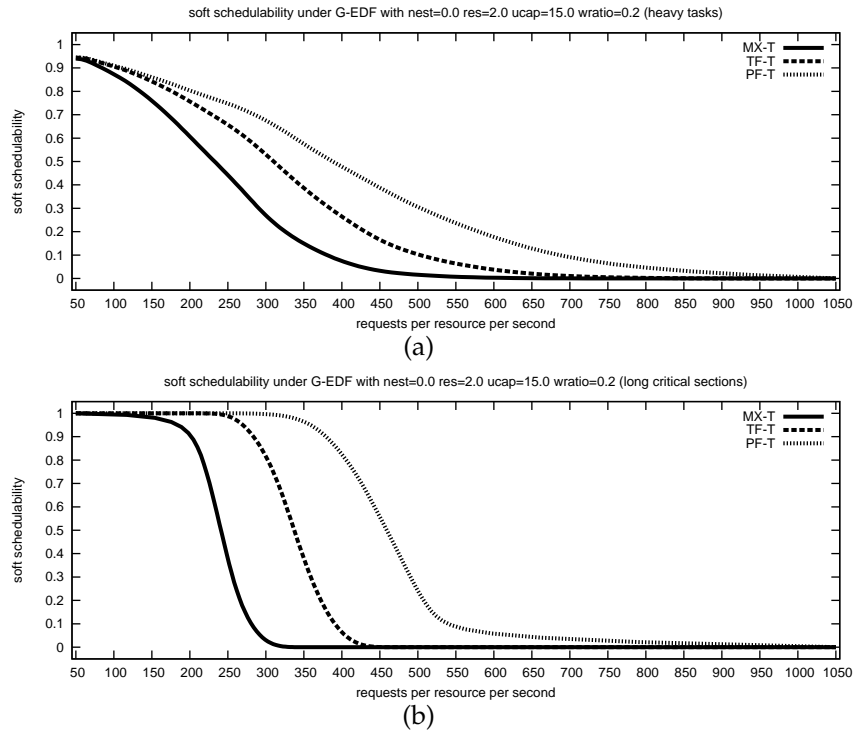


Figure 22: Two variations of the scenario depicted in Figure 13(b), wherein the task set generation procedure was altered by uniformly choosing **(a)** per-task utilizations from $[0.05, 0.95]$ (instead of $[0.1, 0.4]$), and **(b)** critical sections lengths from $[10\mu s, 50\mu s]$ (instead of $[1\mu s, 15\mu s]$).

like to investigate the impact of phase-fairness on throughput-oriented workloads. Another interesting avenue is to explore the relative performance (in terms of schedulability, throughput, and memory requirements) of lock-based RW synchronization and specialized non-blocking synchronization primitives—analogueous to our group’s prior study of mutual exclusion alternatives (Brandenburg et al., 2008b).

References

- Anderson J, Holman P (2000) Efficient pure-buffer algorithms for real-time systems. In: Proceedings of the Seventh International Conference on Real-Time Systems and Applications, pp 57–64
- Anderson J, Kim Y, Herman T (2003) Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing* 16(2-3):75–110
- Anderson J, Bud V, Devi U (2005) An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In: Proceedings of the 17th Euromicro Conference on Real-Time Systems, IEEE, pp 199–208
- Andrews G (1991) Paradigms for process interaction in distributed programs. *ACM Computing Surveys* 23(1):49–90

- Baker T (1991) Stack-based scheduling for realtime processes. *Real-Time Systems* 3(1):67–99
- Baker T (2003) Multiprocessor EDF and deadline monotonic schedulability analysis. In: *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pp 120–129
- Baker T, Baruah S (2007) Schedulability analysis of multiprocessor sporadic task systems. In: Son SH, Lee I, Leung JY (eds) *Handbook of Real-Time and Embedded Systems*, Chapman Hall/CRC, Boca Raton, Florida
- Baruah S (2007) Techniques for multiprocessor global schedulability analysis. In: *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pp 119–128
- Bertogna M, Cirinei M, Lipari G (2005) Improved schedulability analysis of edf on multiprocessor platforms. In: *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pp 209–218
- Bertogna M, Cirinei M, Lipari G (2009) Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on Parallel and Distributed Systems* 20(4):553–566
- Block A, Leontyev H, Brandenburg B, Anderson J (2007) A flexible real-time locking protocol for multiprocessors. In: *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp 47–57
- Brandenburg B, Anderson J (2007) Feather-trace: A light-weight event tracing toolkit. In: *Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pp 20–27
- Brandenburg B, Anderson J (2008) A comparison of the M-PCP, D-PCP, and FMLP on LITMUS^{RT}. In: *Proceedings of the 12th International Conference On Principles Of Distributed Systems*, pp 105–124
- Brandenburg B, Block A, Calandrino J, Devi U, Leontyev H, Anderson J (2007) LITMUS^{RT}: A status report. In: *Proceedings of the 9th Real-Time Linux Workshop*, pp 107–123
- Brandenburg B, Calandrino J, Anderson J (2008a) On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In: *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pp 157–169
- Brandenburg B, Calandrino J, Block A, Leontyev H, Anderson J (2008b) Synchronization on real-time multiprocessors: To block or not to block, to suspend or spin? In: *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp 342–353
- Brandenburg B, Leontyev H, Anderson J (2009) Accounting for interrupts in multiprocessor real-time systems. In: *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp 273–283

- Calandrino J, Leontyev H, Block A, Devi U, Anderson J (2006) LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In: Proceedings of the 27th IEEE Real-Time Systems Symposium, pp 111–123
- Calandrino J, Anderson J, Baumberger D (2007) A hybrid real-time scheduling approach for large-scale multicore platforms. In: Proceedings of the 19th Euromicro Conference on Real-Time Systems, pp 247–256
- Courtois P, Heymans F, Parnas D (1971) Concurrent control with “readers” and “writers”. *Communications of the ACM* 14(10):667–668
- Devi U, Anderson J (2008) Tardiness bounds under global EDF scheduling on a multiprocessor. *Real-Time Systems* 38(2):133–189
- Goossens J, Funk S, Baruah S (2003) Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems* 25(2-3):187–205
- Gore P, Pyarali I, Gill C, Schmidt D (2004) The design and performance of a real-time notification service. In: Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium, pp 112–120
- Hsieh W, Weihl W (1992) Scalable reader-writer locks for parallel systems. In: Proceedings of the 6th International Parallel Processing Symposium, pp 656–659
- Krieger O, Stumm M, Unrau R, Hanna J (1993) A fair fast scalable reader-writer lock. In: Proceedings of the 1993 International Conference on Parallel Processing, pp 201–204
- Leontyev H, Anderson J (2007) Generalized tardiness bounds for global multiprocessor scheduling. In: Proceedings of the 28th IEEE Real-Time Systems Symposium, pp 413–422
- Li Q, Yao C (2003) *Real-Time Concepts for Embedded Systems*. CMP Books
- Liu C, Layland J (1973) Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM* 30:46–61
- Liu J (2000) *Real-Time Systems*. Prentice Hall
- McKenney PE (1996) Selecting locking primitives for parallel programming. *Communications of the ACM* 39(10):75–82
- Mellor-Crummey J, Scott M (1991a) Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems* 9(1):21–65
- Mellor-Crummey J, Scott M (1991b) Scalable reader-writer synchronization for shared-memory multiprocessors. In: Proceedings of the 3rd ACM SIGPLAN symposium on principles and practice of parallel programming, pp 106–113
- Musial M, Remuß V, Deeg C, Hommel G (2006) Embedded system architecture of the second generation autonomous unmanned aerial vehicle MARVIN MARK II. In: Proceedings of the 7th International Workshop on Embedded Systems-Modeling, Technology and Applications, pp 101–110

Rajkumar R (1991) Synchronization In Real-Time Systems – A Priority Inheritance Approach. Kluwer Academic Publishers

Reiman M, Wright P (1991) Performance analysis of concurrent-read exclusive-write. In: Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and modeling of computer systems, pp 168–177

A Bounds on Worst-Case Blocking under the RW-FMLP

In this section, we derive bounds on direct blocking under the RW-FMLP for both global and partitioned scheduling, for each of the following group lock choices: task-fair mutex locks (Section A.4), preference RW locks (Section A.5), task-fair RW locks (Section A.7), and phase-fair RW locks (Section A.8). Finally, we bound arrival blocking in terms of the previously derived bounds on direct blocking under both G-EDF and P-EDF (Section A.9).

A.1 Worst-Case Resource Requirements

To implement the RW-FMLP, *a priori* knowledge of all possible resource nesting is fundamentally required to assign each resource to its appropriate resource group (see Section 3), but the frequency and durations of requests are (for the implementation) irrelevant and may be unknown.

However, to bound worst-case blocking, additional limits on task requirements must be known in advance. In particular, we require for each task T_x and each group g a bound on the maximum number of times that a given number of consecutive jobs of T_x issue outermost write (resp., read) requests for resources in g , and a bound on the maximum duration of each request. This is formalized by the concept of a “generic request sequence,” which is a set of write (resp., read) requests that limits the number and duration of the write (resp., read) requests issued by any k consecutive jobs of T_x (in any schedule).¹⁸

Definition 2. Let W^y denote the set of all write requests for group g issued by the k consecutive jobs $T_x^y, \dots, T_x^{y+k-1}$ in some actual schedule (for $k \geq 1$).

The generic write request sequence, denoted $\mathbf{writes}(T_x, g, k)$, for k consecutive jobs of T_x for resource group g is a set of generic¹⁹ write requests such that both

- for any such W^y , one can choose a set of generic requests $G^y \subseteq \mathbf{writes}(T_x, g, k)$ such that $|W^y| = |G^y|$ and

$$\sum_{W \in W^y} \|W\| \leq \sum_{W \in G^y} \|W\|$$

(recall that $\|W\|$ denotes the length of W , and that $|W^y|$ denotes the cardinality of W^y), and

¹⁸The concept of a generic request sequence is similar to the notion of a *demand bound function* in schedulability analysis (e.g., see (Baruah, 2007)) in the sense that both are used to characterize maximum resource requirements.

¹⁹A *generic* request is simply a request that is defined for analysis purposes.

- $k \leq l \Rightarrow \text{writes}(T_x, g, k) \subseteq \text{writes}(T_x, g, l)$, i.e., generic write request sequences are inclusive.

We analogously let $\text{reads}(T_x, g, k)$ denote the generic read request sequence for k consecutive jobs of T_x .

In general, determining exact definitions of $\text{reads}(T_x, g, k)$ and $\text{writes}(T_x, g, k)$ for each T_x , group g , and $k \geq 1$ may require deep control flow and worst-case execution-time analysis of T_x 's implementation, which is beyond the scope of this paper. As a simple approximation, one could determine the set of all resources requested by at least one job of T_x and then assume that every job of T_x requests all such resources. This, however, can be very pessimistic. Instead, we assume a simplified request model (defined below) that applies the concepts of the sporadic task model to resource requests.

Definition 3. We let $W(T_x, g)$ (resp., $R(T_x, g)$) denote the maximum number of write (resp., read) requests for group g issued by any job of T_x .

Definition 4. Jobs of a task T_x create a sporadic request pattern iff the following two conditions hold for each resource group g for which jobs of T_x issue requests:

- there exist $W(T_x, g)$ pairs $(\text{wrp}_{x,w}^g, \text{wrp}_{x,w}^g)$, where $\text{wrp}_{x,w}^g > 0$ is a request duration and $\text{wrp}_{x,w}^g \geq 1$ is a request period, such that

$$\text{writes}(T_x, g, k) \triangleq \left\{ \mathcal{W}_{x,w}^v \mid 1 \leq w \leq W(T_x, g) \wedge 1 \leq v \leq \left\lceil \frac{k}{\text{wrp}_{x,w}^g} \right\rceil \right\},$$

where $|\mathcal{W}_{x,w}^v| = \text{wrp}_{x,w}^g$; and, similarly,

- there exist $R(T_x, g)$ pairs $(\text{rrp}_{x,r}^g, \text{rrp}_{x,r}^g)$, where $\text{rrp}_{x,r}^g > 0$ and $\text{rrp}_{x,r}^g \geq 1$, such that

$$\text{reads}(T_x, g, k) \triangleq \left\{ \mathcal{R}_{x,r}^v \mid 1 \leq r \leq R(T_x, g) \wedge 1 \leq v \leq \left\lceil \frac{k}{\text{rrp}_{x,r}^g} \right\rceil \right\},$$

where $|\mathcal{R}_{x,r}^v| = \text{rrp}_{x,r}^g$.

For example, suppose a monitor application M reads a sensor \mathcal{L}_s (in group g_1) at a rate of ten samples per second ($\text{p}(T_m) = 100ms$) and updates a shared variable \mathcal{L}_a (in group g_2) storing a history of average readings once every two seconds. Suppose reading sensor \mathcal{L}_s takes at most 1ms and updating variable \mathcal{L}_a takes at most 10ms.

This application can be modeled as a sporadic task T_M with $\text{p}(T_M) = 100ms$, where every job of T_m issues a read request for g_1 and every 20th job of T_M issues a write request for g_2 . In this case, assuming that every job of T_M issues a write request when analyzing contention for g_2 would be unnecessarily pessimistic. Instead, the generic request sequences (both read and write) can be accurately described by a sporadic request pattern for the two parameter pairs $(\text{rrp}_{x,1}^{g_1} = 1ms, \text{rrp}_{x,1}^{g_1} = 1)$, and $(\text{wrp}_{x,1}^{g_2} = 10ms, \text{wrp}_{x,1}^{g_2} = 20)$.

Throughout this paper, we focus on tasks with sporadic request patterns. However, note that the analysis presented in this appendix applies to any definition of $\text{reads}(T_x, g, k)$ and $\text{writes}(T_x, g, k)$ as long as Definition 2 is satisfied. This allows tighter bounds on blocking to be derived if additional information on the frequency of resource requests is available.

A.2 Approach

In the following, we derive blocking terms for an arbitrary job T_i^j with respect to an arbitrary fixed schedule. Since T_i^j and the schedule are arbitrary, this analysis upper bounds worst-case blocking for any job under any schedule.

Definition 5. A request \mathcal{X} for a resource in group g directly blocks T_i^j if T_i^j issues a request \mathcal{X}' for some resource also in g , \mathcal{X}' is issued at time t_0 and is satisfied at time t_1 , where $t_1 > t_0$, \mathcal{X} completes at time t_4 , and $t_4 \in [t_0, t_1]$. Let t_3 denote the time that \mathcal{X} is satisfied. Then \mathcal{X} directly blocks T_i^j during the interval $[\max(t_0, t_3), t_4]$.

Under locks with strong progress guarantees, *i.e.*, task-fair and phase-fair locks, the maximum number of times that a job T_i^j is directly blocked can be bounded by the minimum of both the number of times that T_i^j issues outermost requests and the number of times that remote jobs issue interfering requests. Under locks that allow starvation, *i.e.*, preference locks, the maximum number of times that a job T_i^j is directly blocked can be bounded by the number of times that other jobs issue interfering requests.

Since groups are independent, analysis can be done on a group-by-group basis. Given a bound on the maximum duration of direct blocking incurred by one job of T_i due to requests issued for resources in resource group g , denoted $dbg(T_i, g)$, a bound on total direct blocking is given by the sum of the bounds on blocking for each group that T_i accesses.

Under partitioned scheduling, we assume that the task set has been successfully partitioned prior to computing blocking terms and let $P(T_i)$ denote the processor $(1, \dots, m)$ to which T_i has been assigned. For notational convenience, we assume that $\max(\emptyset) = 0$. In this section, all considered requests are presumed to be outermost unless otherwise noted, as only outermost requests can block under the RW-FMLP. All discussed sets are finite unless noted otherwise.

A.3 Bounding Interference

In this subsection, we characterize the “worst-case interference” that T_i^j may encounter while it is pending in terms of generic requests and use that to bound the duration of actual blocking in the fixed schedule. From a high-level perspective, this involves three steps:

- first, we determine the maximum number of jobs that can execute while T_i^j is pending for each T_x that shares resources with T_i ,
- then we apply Definition 2 assuming this number of jobs of T_x , and
- finally we show how to bound maximum blocking with the “worst-case interference.”

The last step will let us express an upper bound on blocking in terms of generic requests, which are known *a priori*.

Lemma 1. *At most*

$$\maxjobs(T_x, t) \triangleq \left\lceil \frac{t + r(T_x)}{p(T_x)} \right\rceil$$

distinct jobs of a task T_x can execute in any interval of length t .

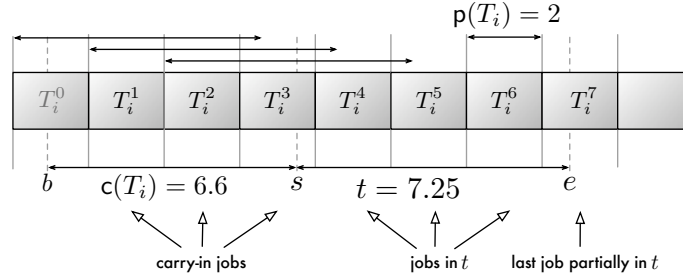


Figure 23: An illustration of Lemma 1. At most seven jobs of T_x with $\mathbf{p}(T_x) = 2$ and $\mathbf{r}(T_x) = 6.6$ can be pending in any interval $[s, e)$ of length $e - s = t = 7.25$ (illustration drawn to scale). Only jobs released at or after $b = s - \mathbf{r}(T_i)$ and before e can be pending in $[s, e]$. In the worst case (i.e., with periodic arrivals and jobs completing as late as possible), jobs T_i^1 , T_i^2 , and T_i^3 carry execution into $[s, e]$. Jobs T_i^4 , T_i^5 , and T_i^6 are released and complete in $[s, e]$, and T_i^7 is released before e . Note that moving the start of the interval s to an earlier point such that $\mathbf{a}(T_i^0) \geq b$ causes the last counted job T_i^7 to no longer be pending within $[s, e)$, i.e., $e < \mathbf{a}(T_i^7)$. Hence, $\maxjobs(T_x, 7.25) = \lceil \frac{13.85}{2.0} \rceil = 7$.

Proof. By contradiction (see Figure 23 for a visual example). Suppose that there exists an interval $[t_0, t_0 + t)$ of length $t \geq 0$ in which $k \in \mathbb{N}$ jobs of T_i execute such that

$$k \geq \left\lceil \frac{t + \mathbf{r}(T_x)}{\mathbf{p}(T_x)} \right\rceil + 1. \quad (1)$$

Let T_x^a denote the first and T_x^z the last job of T_x to execute in $[t_0, t_0 + t)$, where $z = a + k - 1$ (note that $T_i^a \neq T_i^z$ since $\mathbf{r}(T_x) \geq \mathbf{e}(T_x) > 0$ and hence $k \geq 2$). In order for a job to execute in $[t_0, t_0 + t)$, it must be pending some time in $[t_0, t_0 + t)$, i.e.,

$$t_0 - \mathbf{r}(T_x) \leq \mathbf{a}(T_x^a) \text{ and} \quad (2)$$

$$\mathbf{a}(T_x^z) < t_0 + t. \quad (3)$$

Further, since job releases are separated by at least one period, it follows that

$$\mathbf{a}(T_x^a) + (k - 1) \cdot \mathbf{p}(T_x) \leq \mathbf{a}(T_x^z). \quad (4)$$

By substituting Inequalities (2) and (3) into Inequality (4), we obtain

$$t_0 - \mathbf{r}(T_i) + (k - 1) \cdot \mathbf{p}(T_i) \leq t_0 + t,$$

which, by re-arranging, yields

$$k < \frac{t + \mathbf{r}(T_i)}{\mathbf{p}(T_i)} + 1 \leq \left\lceil \frac{t + \mathbf{r}(T_i)}{\mathbf{p}(T_i)} \right\rceil + 1$$

This contradicts Inequality (1) above. □

Next, we define T_i 's "competition," i.e., the set of tasks that have jobs that issue blocking requests.

Definition 6. A task T_x , $x \neq i$, competes with T_i for resource group g if some jobs of both tasks request resources in g , i.e., if both $R(T_x, g) + W(T_x, g) > 0$ and $R(T_i, g) + W(T_i, g) > 0$. The competition $C(T_i, g)$ denotes the set of all tasks that compete with T_i for g .

In the second step, we obtain a set of generic requests that we will subsequently use to upper bound the actual blocking duration.

Definition 7. The worst-case write interference $wif(T_x, g, t)$ of task T_x with respect to group g during any interval of length t is the set of generic requests

$$wif(T_x, g, t) \triangleq \mathbf{writes}(T_x, g, \maxjobs(T_x, t)).$$

We similarly define

$$rif(T_x, g, t) \triangleq \mathbf{reads}(T_x, g, \maxjobs(T_x, t))$$

and

$$xif(T_x, g, t) \triangleq wif(T_x, g, t) \cup rif(T_x, g, t)$$

to denote the worst-case read interference and the worst-case request interference.

Next, we define three convenience functions over sets of requests, which serve to simplify the expression of “aggregate interference” (see Def. 10 below).

Definition 8. Given a set of requests S , we let S_k denote the k th longest request in S , where $1 \leq k \leq |S|$ (with ties broken in an arbitrary but consistent fashion). Formally, $1 \leq k \leq l \leq |S| \Rightarrow \|S_k\| \geq \|S_l\|$.

Definition 9. Given a set of requests S , we denote the set of the l longest requests in A and their total duration as

$$\begin{aligned} top(l, S) &\triangleq \{S_k \mid k \in \{1, \dots, \min(l, |S|)\}\}, \text{ and} \\ total(l, S) &\triangleq \sum_{\mathcal{X} \in top(l, S)} \|\mathcal{X}\|. \end{aligned}$$

Definition 10. The aggregate worst-case write interference of T_i 's competition with respect to a resource group g over any interval of length t and subject to an interference limit l is given by

$$cwifs(T_i, g, t, l) = \bigcup_{T_x \in C(T_i, g)} top(l, wif(T_x, g, t)).$$

Similarly, we define

$$crifs(T_i, g, t, l) = \bigcup_{T_x \in C(T_i, g)} top(l, rif(T_x, g, t))$$

and

$$cxifs(T_i, g, t, l) = \bigcup_{T_x \in C(T_i, g)} top(l, xif(T_x, g, t))$$

to denote aggregate worst-case read interference and aggregate worst-case request interference.

To this point, no assumptions have been made regarding the scheduling policy that created the fixed schedule. Hence, Definition 10 applies to both partitioned and global scheduling in general, and G-EDF and P-EDF in particular. Under partitioned scheduling, a better approximation of aggregate interference can be obtained since the processors on which tasks reside are known in advance.

For example, suppose that all of T_i^j 's competitors (with regard to a given group g) reside on the same processor. Since requests are executed non-preemptively, this allows us to conclude that at most one request issued by a job of a competitor executes at any time. Under group locks with strong fairness guarantees (e.g., FIFO queueing), this can result in significantly reduced worst-case blocking (compared to the worst case under global scheduling). However, Definition 10 is oblivious to partitions and hence cannot reflect such considerations. Thus, we analogously define a partition-aware version of aggregate interference next.

Definition 11. Let $P = \{p \mid p \in \{1, \dots, m\} \wedge p \neq P(T_i)\}$ denote the set of all remote partitions, and let $part(p) = \{T_x \mid T_x \in C(T_i, g) \wedge P(T_x) = p\}$ denote the set of all competitors assigned to processor p .

The partitioned worst-case write interference of T_i 's competition with respect to a resource group g over any interval of length t and subject to an interference limit l is given by

$$pwifs(T_i, g, t, l) = \bigcup_{p \in P} top \left(l, \bigcup_{T_x \in part(p)} wif(T_x, g, t) \right).$$

Similarly, we define

$$pwifs(T_i, g, t, l) = \bigcup_{p \in P} top \left(l, \bigcup_{T_x \in part(p)} rif(T_x, g, t) \right).$$

and

$$pwifs(T_i, g, t, l) = \bigcup_{p \in P} top \left(l, \bigcup_{T_x \in part(p)} xif(T_x, g, t) \right).$$

to denote partitioned worst-case read interference and partitioned worst-case request interference.

To simplify the statement of the bounds, we defined the following short-hand notation.

Definition 12. The worst-case write interference for a resource group g encountered by task T_i over any interval of length t with interference limit l is given by

$$wifs(T_i, g, t, l) = \begin{cases} cwifs(T_i, g, t, l) & \text{under global scheduling} \\ pwifs(T_i, g, t, l) & \text{under partitioned scheduling.} \end{cases}$$

Analogously, we let $rifs(T_i, g, t, l)$ and $xifs(T_i, g, t, l)$ denote the worst-case read interference and worst-case request interference.

A.4 Maximum Direct Blocking under Task-Fair Mutex Locks

Recall that, under task-fair mutex locks, both readers and writers gain exclusive access to resource groups and that all requests are satisfied in FIFO order. Hence, when bounding direct blocking under task-fair mutex locks, the request type (read or write) is irrelevant.

Lemma 2. *Let c denote the number of requests (either read or write) that T_i^j issues for resources in resource group g . Under task-fair mutex locks, jobs of each competitor T_x can block T_i^j 's requests for resources in g with at most c requests.*

Proof. By contradiction. Assume jobs of some task T_x directly block T_i^j with more than c requests. Then at least one request \mathcal{X}^i issued by T_i^j is blocked by at least two requests ($\mathcal{X}_1^x, \mathcal{X}_2^x$) of jobs of T_x . Because both \mathcal{X}_1^x and \mathcal{X}_2^x block \mathcal{X}^i , neither of the two is complete when \mathcal{X}^i is issued, but both are complete by the time that \mathcal{X}^i is satisfied (see Definition 5).

Without loss of generality, assume that \mathcal{X}_1^x is issued prior to \mathcal{X}_2^x . Because jobs are sequential (and because both \mathcal{X}_1^x and \mathcal{X}_2^x are outermost), \mathcal{X}_1^x must complete before \mathcal{X}_2^x is issued. Hence \mathcal{X}_2^x is issued *after* \mathcal{X}^i is issued, but satisfied *before* \mathcal{X}^i is satisfied. This contradicts the assumption that the blocking requests are satisfied in FIFO order. \square

Lemma 3. *Let c denote the number of requests (either read or write) that T_i^j issues for resources in resource group g . Under task-fair mutex locks and under partitioned scheduling, jobs on each remote partition can block T_i^j 's requests for resources in g with at most c requests.*

Proof. Due to the non-preemptive execution of requests, at most one request per remote processor may interfere at any given time (*i.e.*, requests are executed sequentially). Hence, the claim follows analogous to Lemma 2 above. \square

Lemma 4. *Let c denote the number of requests (either read or write) that T_i^j issues for resources in resource group g . Under task-fair mutex locks, T_i^j 's requests for resources in g are blocked by at most $c \cdot (m - 1)$ requests.*

Proof. Consider a request \mathcal{X} issued by T_i^j . At most $m - 1$ other requests can have been issued but not yet be complete at the time that \mathcal{X} is issued because requests are executed non-preemptively. Hence, due to FIFO ordering, each request of T_i^j can be blocked by at most $m - 1$ requests. Consequently, no more than $c \cdot (m - 1)$ requests (and hence phases) block T_i^j in total. \square

Theorem 1. *Let c be the maximum number of requests issued by any job of T_i for resources in group g , and let $t = r(T_i)$. If g is protected by a task-fair mutex lock, then*

$$dbg(T_i, g) \leq total((m - 1) \cdot c, xifs(T_i, g, t, c)). \quad (5)$$

Proof. By Lemmas 2 and 3, each competing task and, under partitioning, each remote processor can block T_i with no more than c interfering requests each. By Lemma 4, each request is directly blocked by at most $(m - 1)$ remote jobs. Hence, at most $(m - 1) \cdot c$ interfering requests can block T_i^j in total. Therefore, the maximum total blocking is bounded by the sum of the durations of the $(m - 1) \cdot c$ longest requests in $xifs(T_i, g, t, c)$. \square

Note that, under global scheduling, (5) yields a bound that is less pessimistic for $c > 1$ than the one previously given in (Block et al., 2007). In the partitioned case, (5) is equivalent to the bound given in (Brandenburg and Anderson, 2008).

A.5 Maximum Direct Blocking under Preference RW Locks

Recall from Section 3.2 that there are two kinds of preference locks. Under a reader preference lock, write requests are only satisfied if no readers are present. Similarly, under a writer preference lock, read requests are only satisfied if no writers are present. In both cases, we assume that write requests are satisfied in FIFO order (with respect to other write requests).

A.5.1 Reader Preference Locks

In Lemmas 5, 6, and 7 below, let $c_{\mathcal{R}}$ denote the number of read requests and $c_{\mathcal{W}}$ the number of write requests that T_i^j issues for resources in resource group g .

Lemma 5. *Under reader preference locks, jobs of each competitor T_x can block T_i^j 's requests for resources in g with at most $c_{\mathcal{R}} + c_{\mathcal{W}}$ write requests. Similarly, under partitioning, jobs on each remote partition can block T_i^j 's requests for resources in g with at most $c_{\mathcal{R}} + c_{\mathcal{W}}$ write requests.*

Proof. Since write requests are not satisfied while a reader is present, at most one (earlier-satisfied) writer phase can block each of T_i^j 's read requests. Since write requests are satisfied in FIFO order, at most one request per competing task can block each of T_i^j 's write requests (analogously to Lemma 2). Similar reasoning applies to the partitioned case. \square

Lemma 6. *Under reader preference locks, if $c_{\mathcal{W}} = 0$, then T_i^j is not blocked by any reader phases.*

Proof. Follows from the definition of reader preference. \square

Lemma 7. *Under reader preference locks, T_i^j 's requests for resources in g are blocked by at most $c_{\mathcal{R}} + (m - 1) \cdot c_{\mathcal{W}}$ write requests.*

Proof. As discussed in Lemma 5 above, each read request is blocked by at most one writer phase. Due to FIFO ordering, each write request of T_i^j is preceded by at most one writer on every other processor (analogously to Lemma 4). \square

Theorem 2. *Let $c_{\mathcal{R}}$ be the maximum number of outermost read requests and let $c_{\mathcal{W}}$ be the maximum number of outermost write requests that any job of task T_i issues for resources in group g , and let $t = r(T_i)$. If g is protected by a reader preference lock, then*

$$dbg(T_i, g) \leq \text{total}(c_{\mathcal{R}} + (m - 1) \cdot c_{\mathcal{W}}, wifs(T_i, g, t, c_{\mathcal{R}} + c_{\mathcal{W}})) + \text{total}(r, xifs(T_i, g, t, r)), \quad (6)$$

where $r = \infty$ if $c_{\mathcal{W}} > 0$; otherwise $r = 0$.

Proof. Lemma 7 limits the number of write requests blocking a job of T_i to $c_{\mathcal{R}} + (m - 1) \cdot c_{\mathcal{W}}$; Lemma 5 limits per-task write interference to $c_{\mathcal{R}} + c_{\mathcal{W}}$. Blocking due to read interference is either zero (if T_i does not issue write requests, by Lemma 6) or limited only by the behavior of competing tasks. The stated bound follows. \square

A.5.2 Writer Preference Locks

In the following Lemmas 8 and 9, let $c_{\mathcal{R}}$ denote the number of read requests and $c_{\mathcal{W}}$ the number of write requests that T_i^j issues for resources in resource group g .

Lemma 8. *Under writer preference locks, if $c_{\mathcal{R}} = 0$, then*

- *jobs of each competitor T_x can block T_i^j 's requests for resources in g with at most $c_{\mathcal{W}}$ requests of any kind;*
- *under partitioning, jobs on each remote partition can block T_i^j 's requests for resources in g with at most $c_{\mathcal{W}}$ requests of any kind;*
- *T_i^j 's requests for resources in g are blocked by at most $c_{\mathcal{W}}$ reader phases; and*
- *T_i^j 's requests for resources in g are blocked by at most $c_{\mathcal{W}} \cdot (m - 1)$ phases (either read or write).*

Proof. Since write requests are satisfied in FIFO order, at most one request per competing task can block each of T_i^j 's write requests (analogously to Lemmas 2 and 4).

Writers are prioritized over readers, thus a write request \mathcal{W} is only blocked by a reader phase if \mathcal{W} is issued during said phase. Hence, at most one reader phase blocks each of T_i^j 's write requests.

A job T_x^y cannot block a single request of T_i^j with both a read request and a write request: after completing its read request, T_x^y 's write request is queued after T_i^j 's write request due to FIFO ordering, and no read request is satisfied while a writer is present. Thus, the number of write requests issued by T_i^j and m limit the number of blocking phases. □

Lemma 9. *Under writer preference locks, if T_i^j is blocked by at most w writer phases, then T_i^j is blocked by at most $c_{\mathcal{W}} + \min(c_{\mathcal{R}}, w)$ reader phases.*

Proof. As before in Lemma 8, a write request can be blocked only once by a reader phase, thus the number of reader phases that block write requests of T_i^j is limited by $c_{\mathcal{W}}$.

Each time that T_i^j issues a read request \mathcal{R} , it can be *transitively* blocked by an earlier reader phase. Suppose a write request is issued just before T_i^j issues \mathcal{R} and is itself blocked by an earlier reader phase: T_i^j transitively incurs blocking while the preceding writer is blocked. Thus, T_i^j may be blocked by up to $c_{\mathcal{R}}$ reader phases.

However, in order for T_i^j to be blocked by a reader phase, a writer must be present. Thus, w also bounds the number of blocking reader phases. □

Theorem 3. *Let $c_{\mathcal{R}}$ be the maximum number of outermost read requests and let $c_{\mathcal{W}}$ be the maximum number of outermost write requests that any job of task T_i issues for resources in group g , and let $t = r(T_i)$. If g is protected by a writer preference lock, then*

$$\text{dbg}(T_i, g) \leq \text{total}(w, W) + \text{total}(x, XR) \quad (7)$$

where, if $c_{\mathcal{R}} = 0$,

$$\begin{aligned} w &= (m - 2) \cdot c_{\mathcal{W}} & x &= c_{\mathcal{W}} \\ W &= wifs(T_i, g, t, c_{\mathcal{W}}) & XR &= xifs(T_i, g, t, c_{\mathcal{W}}) \setminus top(w, W), \end{aligned}$$

and otherwise

$$\begin{aligned} w &= \infty & x &= c_{\mathcal{W}} + \min(c_{\mathcal{R}}, |W|) \\ W &= wifs(T_i, g, t, \infty) & XR &= rifs(T_i, g, t, x). \end{aligned}$$

Proof. By case analysis.

Case $c_{\mathcal{R}} = 0$: If T_i does not issue read requests for resources in g , then a job of T_i is blocked by at most $(m - 1) \cdot c_{\mathcal{W}}$ requests in total and by at most $c_{\mathcal{W}}$ read requests (Lemma 8). Also by Lemma 8, jobs of each remote task and, under partitioning, jobs on each remote processor may block T_i^j with at most $c_{\mathcal{W}}$ write requests each. Hence, the worst-case write interference is given by $wifs(T_i, g, t, c_{\mathcal{W}})$.

By Lemma 8, at most $c_{\mathcal{W}}$ of the $(m - 1) \cdot c_{\mathcal{W}}$ blocking requests in the worst-case scenario may be read requests. Hence, the maximum possible blocking is bounded by the duration of the $(m - 2) \cdot c_{\mathcal{W}}$ longest interfering write requests (if there are that many) and the remaining longest $c_{\mathcal{W}}$ requests of either kind (read or write, but without accounting for the same write request twice).

Case $c_{\mathcal{R}} > 0$: If T_i issues read requests for resources in g , then the number of blocking write requests is only bounded by the behavior of the competing tasks; consequently $w = \infty$ as all possibly interfering write requests must be considered.

By Lemma 9, each of T_i 's requests (either read or write) may be blocked by at most $x = c_{\mathcal{W}} + \min(c_{\mathcal{R}}, |W|)$ reader phases; consequently the worst-case read interference is given by $rifs(T_i, g, t, x)$. □

A.6 Reader Parallelism under Task-Fair and Phase-Fair RW Locks

In this subsection, we establish two lemmas that characterize reader parallelism under fair RW locks (both task-fair and phase-fair). They formalize the intuition that a reader phase can only transitively block a read request if said phase is “assisted” by an also-blocking writer phase.

Lemma 10. *Let $c_{\mathcal{W}}$ denote the number of write requests that a job T_i^j issues for resources in group g , and let*

- w denote the number of blocking writer phases,
- r denote the number of blocking reader phases,
- b_w denote the total number of blocking phases (either read or write).

If g is protected by a fair RW lock (either task-fair or phase-fair), then

$$b_w \leq 2w + c_{\mathcal{W}} \text{ and} \qquad r \leq w + c_{\mathcal{W}}.$$

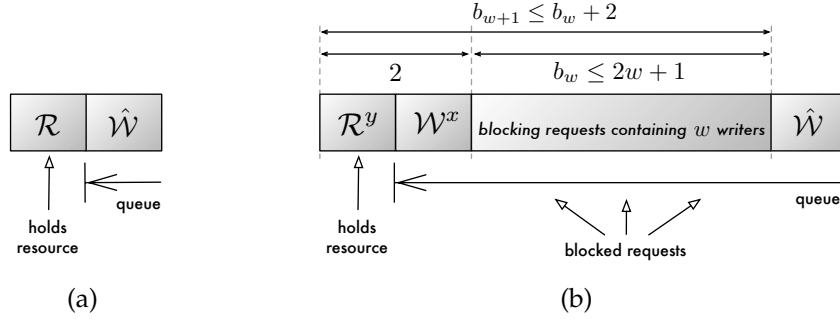


Figure 24: Illustration of Lemma 10 for the case $c_{\mathcal{W}} = 1$ wherein a job T_i^j issues only one write request \hat{W} . **(a)** Base case $w = 0$: at most one reader phase can block \hat{W} . **(b)** Induction step: allowing one additional write request \mathcal{W}^x increases the maximum number of blocking phases by at most two.

Proof. By induction over w . Note that $b_w = w + r$.

Base case: $w = 0$. If there are no blocking writer phases, then T_i^j 's read requests (if any) do not incur blocking at all, and, due to FIFO ordering in task-fair locks and from Properties PF1 and PF4 in phase-fair locks, T_i^j 's write requests (if any) can only be blocked by one reader phase each. Hence, $r \leq c_{\mathcal{W}}$, and thus $r = r + w = b_w \leq 2w + c_{\mathcal{W}}$. This is illustrated in inset (a) of Figure 24.

Induction step $w \rightarrow w + 1$: Adding one blocking write request \mathcal{W}^x (i.e., one writer phase) increases the number of phases blocking T_i^j by at most two: firstly, \mathcal{W}^x directly blocks T_i^j once, and, secondly, \mathcal{W}^x can itself be blocked by an additional read request \mathcal{R}^y (and thus one reader phase), which then transitively also blocks T_i^j . Thus, $b_{w+1} \leq b_w + 2$. This is illustrated in inset (b) of Figure 24. By the induction hypothesis, $b_w \leq 2w + c_{\mathcal{W}}$, thus

$$b_{w+1} \leq b_w + 2 \leq (2w + c_{\mathcal{W}}) + 2 = 2(w + 1) + c_{\mathcal{W}}.$$

Hence, $b_w \leq 2w + c_{\mathcal{W}}$ for all $w \in \mathbb{N}$. Since $b_w = w + r$, this implies $r \leq w + c_{\mathcal{W}}$. \square

Lemma 10 bounds b_w for a known w . Next, we derive a bound on r given a bound on b_w .

Lemma 11. Let $c_{\mathcal{W}}$ denote the number of write requests that a job T_i^j issues for resources in group g , and define w , r , and b_w as in Lemma 10 above. If there exists a bound $a \in \mathbb{N}$ such that $b_w \leq a$, then

$$r \leq \left\lfloor \frac{a + c_{\mathcal{W}}}{2} \right\rfloor.$$

Proof. We first show $r \leq \frac{a + c_{\mathcal{W}}}{2}$ by contradiction. Suppose

$$\frac{a + c_{\mathcal{W}}}{2} < r.$$

Since $b_w = w + r$ and thus $r \leq a - w$, the stated inequality implies

$$\frac{a + c_{\mathcal{W}}}{2} < a - w$$

and thus

$$w < a - \frac{a + c_W}{2} = \frac{a - c_W}{2}.$$

By solving for a we obtain

$$a > 2w + c_W \geq b_w,$$

where the last inequality follows from Lemma 10. This contradicts the assumption $b_w \leq a$, and thus, $r \leq \frac{a+c_W}{2}$ holds. Since $r \in \mathbb{N}$, the stated inequality follows. \square

A.7 Maximum Direct Blocking under Task-Fair RW Locks

Recall that under task-fair RW locks, writers gain exclusive access to resource groups, whereas *consecutive* readers may access resource groups concurrently. Consequently, a writer that is directly blocked by a reader phase may cause the reader phase to transitively block later-arriving readers.

Lemma 12. *Let $c_{\mathcal{R}}$ denote the number of read requests and $c_{\mathcal{W}}$ the number of write requests that T_i^j issues for resources in resource group g . Under task-fair RW locks,*

- *jobs of each competitor T_x can block T_i^j 's requests for resources in g with at most $c_{\mathcal{W}} + c_{\mathcal{R}}$ requests of any kind;*
- *under partitioning, jobs on each remote partition can block T_i^j 's requests for resources in g with at most $c_{\mathcal{W}} + c_{\mathcal{R}}$ requests of any kind; and*
- *T_i^j 's requests for resources in g are blocked by at most $(c_{\mathcal{W}} + c_{\mathcal{R}}) \cdot (m - 1)$ phases (either read or write).*

Proof. Follows analogously to Lemmas 2, 3, and 4 since all requests are satisfied in strict FIFO order. \square

Theorem 4. *Let $c_{\mathcal{R}}$ denote the maximum number of read requests and let $c_{\mathcal{W}}$ denote the maximum number of write requests that any job of task T_i issues for resources in group g , and let $t = r(T_i)$. If g is protected by a task-fair RW lock, then*

$$dbg(T_i, g) \leq \min\left(\text{total}(a, X), \text{total}(a - r, W) + \text{total}(r, X \setminus \text{longest}(w, W))\right),$$

where

$$\begin{aligned} W &= wifs(T_i, g, t, c_{\mathcal{R}} + c_{\mathcal{W}}) & X &= xifs(T_i, g, t, c_{\mathcal{R}} + c_{\mathcal{W}}) \\ a &= \min((m - 1) \cdot (c_{\mathcal{R}} + c_{\mathcal{W}}), |W| \cdot 2 + c_{\mathcal{W}}) & r &= \left\lfloor \frac{a + c_{\mathcal{W}}}{2} \right\rfloor. \end{aligned}$$

Proof. By Lemma 12, each competing task and, under partitioning, the jobs of the tasks on each remote partition may be blocked by at most $c_{\mathcal{R}} + c_{\mathcal{W}}$ requests. Hence, the worst-case write and request interference is given by $wifs(T_i, g, t, c_{\mathcal{R}} + c_{\mathcal{W}})$ and $xifs(T_i, g, t, c_{\mathcal{R}} + c_{\mathcal{W}})$ (resp.). The total number of phases that block T_i^j is at most $(m - 1) \cdot (c_{\mathcal{R}} + c_{\mathcal{W}})$ by Lemma 12, and at most at most $|W| \cdot 2 + c_{\mathcal{W}}$ by Lemma 10, thus a is an upper bound on the total number of blocking phases. Hence, a bound on $dbg(T_i, g)$ is given by the sum of the a longest requests in X .

Further, by Lemma 11, at most $r = \left\lfloor \frac{a + c_{\mathcal{W}}}{2} \right\rfloor$ blocking phases are reader phases. This yields a second bound: the $a - r$ longest competing write requests and the r longest requests of either kind (read or write, without accounting for the same write request twice) also bound $dbg(T_i, g)$. \square

Two bounds on $db(T_i, g)$ are given because either bound can overestimate the true maximum blocking time in certain cases. For example, if write requests are generally short and there are many long read requests, then the sum of the a longest requests of either kind can be unnecessarily pessimistic as it does not differentiate between read and write requests, *i.e.*, the bound could equal the sum of the $(m - 1) \cdot (c_{\mathcal{W}} + c_{\mathcal{R}})$ longest read requests, even though that is clearly not a feasible scenario. Similarly, the bound based on Lemma 11 can overestimate total blocking time since it may account for more than $c_{\mathcal{R}} + c_{\mathcal{W}}$ requests per task, which is also not possible (Lemma 12). Hence, it is beneficial to compute both bounds.

A.8 Maximum Direct Blocking under Phase-Fair RW Locks

Recall that, under phase-fair locks, readers and writers are ordered only with regard to phases, but progress is guaranteed because reader and writer phases alternate (Properties PF1 and PF4). Writers gain access in strict FIFO order with respect to other writers (Property PF2). At the beginning of a reader phase, all currently-blocked readers gain concurrent access (Property PF3). Hence, a reader may gain access to a resource group prior to an earlier-arrived writer. This “skipping ahead” of readers expedites read requests, but increases the number of times that a given task may issue blocking read requests since now a single write request can be blocked by jobs of the same task multiple times. This tradeoff is a conscious design decision based on the expectation that RW locks are only employed when read requests significantly outnumber write requests.

In Lemmas 8 and 9 below, let $c_{\mathcal{R}}$ denote the number of read requests and $c_{\mathcal{W}}$ the number of write requests that T_i^j issues for resources in resource group g .

Lemma 13. *Under phase-fair RW locks,*

- *jobs of each competitor T_x can block T_i^j 's requests for resources in g with at most $c_{\mathcal{R}} + c_{\mathcal{W}}$ write requests;*
- *under partitioning, jobs on each remote partition can block T_i^j 's requests for resources in g with at most $c_{\mathcal{R}} + c_{\mathcal{W}}$ write requests; and*
- *T_i^j 's requests for resources in g are blocked by at most $c_{\mathcal{R}} + c_{\mathcal{W}} \cdot (m - 1)$ writer phases.*

Proof. Since all blocked read requests are satisfied at the beginning of a reader phase, and since reader and writer phases alternate, at most one writer phase can block each of T_i^j 's $c_{\mathcal{R}}$ read requests. Since write requests are satisfied in FIFO order, each of T_i^j 's $c_{\mathcal{W}}$ write requests can be blocked by up to $(m - 1)$ writer phases (analogously to Lemma 4).

Due to FIFO ordering of writes, jobs of each competing task may block each of T_i^j 's write and read requests at most once (analogously to Lemma 2). \square

Lemma 14. *Under phase-fair RW locks,*

- *jobs of each competitor T_x can block T_i^j 's requests for resources in g with at most $c_{\mathcal{R}} + c_{\mathcal{W}} \cdot (m - 1)$ read requests;*
- *under partitioning, jobs on each remote partition can block T_i^j 's requests for resources in g with at most $c_{\mathcal{R}} + c_{\mathcal{W}} \cdot (m - 1)$ read requests; and*
- *T_i^j 's requests for resources in g are blocked by at most $c_{\mathcal{R}} + c_{\mathcal{W}} \cdot (m - 1)$ reader phases.*

Proof. A read request \mathcal{R} issued by T_i^j can be blocked by a reader phase only if \mathcal{R} is issued during said reader phase and an earlier-issued write request is currently blocked. By Property PF4, \mathcal{R} is not satisfied until the beginning of the next reader phase. Thus, each of T_i^j 's read requests is blocked by at most one reader phase.

Once a write request \mathcal{W} issued by T_i^j is at the head of the writer queue, it is blocked by at most one reader phase since reader and writer phases alternate. However, in the worst case, there are $m - 1$ writers ahead of \mathcal{W} . Since a reader phase can occur between any two consecutive writer phases, \mathcal{W} may be transitively blocked by up to $m - 2$ reader phases, for a total of at most $m - 1$ reader phases per request.

Since no ordering among read requests is enforced, each competing task may block T_i^j with one request during each blocking reader phase. \square

Theorem 5. *Let $c_{\mathcal{R}}$ denote the maximum number of read requests and let $c_{\mathcal{W}}$ denote the maximum number of write requests that any job of task T_i issues for resources in group g , and let $t = r(T_i)$. If g is protected by a phase-fair RW lock, then*

$$dbg(T_i, g) \leq total(c_{\mathcal{R}} + (m - 1) \cdot c_{\mathcal{W}}, W) + total(r, R)$$

where

$$\begin{aligned} W &= wifs(T_i, g, t, c_{\mathcal{R}} + c_{\mathcal{W}}) & r &= \min(|W| + c_{\mathcal{W}}, c_{\mathcal{R}} + (m - 1) \cdot c_{\mathcal{W}}) \\ R &= rifs(T_i, g, t, r). \end{aligned}$$

Proof. The worst-case write interference is given by $wifs(T_i, g, t, c_{\mathcal{R}} + c_{\mathcal{W}})$ since, by Lemma 13, per-task and, under partitioning, per-processor write interference is limited by $c_{\mathcal{R}} + c_{\mathcal{W}}$. Also by Lemma 13, T_i^j is blocked by at most $c_{\mathcal{R}} + (m - 1) \cdot c_{\mathcal{W}}$ write requests in total, and hence maximum blocking due to competing write requests cannot exceed $total(c_{\mathcal{R}} + (m - 1) \cdot c_{\mathcal{W}}, W)$.

The total number of blocking read requests r is bounded both by $|W| + c_{\mathcal{W}}$ (due to Lemma 11) and by $c_{\mathcal{R}} + (m - 1) \cdot c_{\mathcal{W}}$ (due to Lemma 14). Hence, $R = rifs(T_i, g, t, r)$ upper-bounds the worst-case read interference, and thus maximum blocking due to competing read requests cannot exceed $total(r, R)$. \square

This concludes the derivation of bounds on direct blocking for each of the considered group lock choices. Next, we show how a bound on arrival blocking can be expressed in terms of direct blocking.

A.9 Maximum Arrival Blocking under G-EDF and P-EDF

Arrival blocking occurs when a newly-released job has a sufficiently-high priority to execute immediately on a processor p but cannot be scheduled because a lower-priority job is executing non-preemptively on p . If jobs can suspend, then they can also incur arrival blocking when they resume—however, in the context of this paper, jobs are assumed to not suspend their execution, and thus only job releases must be considered.

Lemma 15. *Under EDF (either partitioned or global), if T_x causes T_i to incur arrival blocking (and if jobs of T_i do not self-suspend), then $\mathbf{d}(T_x) > \mathbf{d}(T_i)$.*

Proof. Let T_x^y denote the job that should be preempted but is non-preemptive when T_i^j arrives at time $t = \mathbf{a}(T_i^j)$, i.e., T_i^j has an earlier absolute deadline $d_i^j = t + \mathbf{d}(T_i)$. Since T_x^y was already executing at time t , $\mathbf{a}(T_x^y) \leq t$. Hence, T_x^y 's absolute deadline d_x^y is at most $t + \mathbf{d}(T_x)$. By assumption, $d_i^j < d_x^y$ and thus $t + \mathbf{d}(T_i) < t + \mathbf{d}(T_x) \Leftrightarrow \mathbf{d}(T_i) < \mathbf{d}(T_x)$. \square

Lemma 16. *The set of tasks that may cause T_i to incur arrival blocking is given by*

$$A(T_i) = \begin{cases} \{T_x \mid x \in \{1, \dots, n\} \wedge \mathbf{p}(T_x) > \mathbf{p}(T_i)\} & \text{under G-EDF,} \\ \{T_x \mid x \in \{1, \dots, n\} \wedge \mathbf{p}(T_x) > \mathbf{p}(T_i) \wedge \mathbf{P}(T_x) = \mathbf{P}(T_i)\} & \text{under P-EDF.} \end{cases} \quad (8)$$

Proof. Follows directly from Lemma 15 and the fact that, under partitioning, only local tasks can cause arrival blocking. \square

Arrival blocking is bounded by the length of the longest non-preemptive section executed by any job with lower priority. We assume that jobs become (briefly) preemptive between consecutive outermost requests,²⁰ thus a bound on the length of non-preemptive sections is given by the maximum blocking time incurred by a single request and the duration of the request itself.

Lemma 17. *Let \mathcal{X} denote a request (either read or write) of a job T_x^y for a resource in group g . A per-request bound on maximum direct blocking, denoted $rb(\mathcal{X})$, can be obtained by computing $dbg(T_x^y, g)$ under the assumption that \mathcal{X} is the only request issued by T_x^y .*

Proof. Since requests of one job are independent under the RW-FMLP, the worst-case blocking that \mathcal{X} can incur is no more than the worst-case blocking that T_x^y can incur if it issues only \mathcal{X} . \square

Theorem 6. *Let $G(T_x)$ denote the set of resource groups accessed by jobs of task T_x . Worst-case arrival blocking incurred by any job of task T_i is bounded by*

$$ab(T_i) = \max \left\{ \|\mathcal{X}\| + rb(\mathcal{X}) \mid \mathcal{X} \in \bigcup_{T_x \in A(T_i)} \bigcup_{g \in G(T_x)} xif(T_x, g, \mathbf{r}(T_i)) \right\}. \quad (9)$$

²⁰For example, in OS kernels that disable preemptions, it is common for long-running code paths to contain *preemption points* between critical sections, at which the scheduled job checks for the presence of higher-priority jobs.

Proof. Follows from the preceding discussion. □

Note that only the bound on arrival blocking depends on scheduling priority, whereas the bounds on direct blocking differentiate only between global and partitioned scheduling, but not the actual scheduling policy. The analysis presented in this section can thus be readily applied to other real-time scheduling policies (such as P-SP) with only minor adjustments to the bound on arrival blocking (as long as requests are executed non-preemptively).