Reader-Writer Synchronization for Shared-Memory Multiprocessor Real-Time Systems*

Björn B. Brandenburg and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

Reader preference, writer preference, and task-fair reader-writer locks are shown to cause undue blocking in multiprocessor real-time systems. A new phase-fair reader-writer lock is proposed as an alternative that significantly reduces worst-case blocking for readers and an efficient local-spin implementation is provided. Both task- and phase-fair locks are evaluated and contrasted to mutex locks in terms of hard and soft real-time schedulability under consideration of runtime overheads on a multicore computer.

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 [11, 14, 15, 17] 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 data). As an example for the former, consider a robot such as TU Berlin's autonomous helicopter Marvin [29]: 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 targeting, and communication modules. An example of rarely-changing shared data occurs in work of Gore *et al.* [21], 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." [21]

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 starvation have been suggested to practitioners without much concern for schedulability [24]. 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 [18]: 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 large parallel systems, Hsieh and Weihl proposed a semaphore-based RW lock wherein the lock state is distributed across processors [22]. 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 [28]. 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.* [23]. A probabilistic performance analysis of task-fair RW locks wherein reader arrivals are modeled as a Poisson process was conducted by Reiman and Wright [31].

In work on uniprocessor real-time systems, two relevant suspension-based protocols have been proposed: Baker's stack resource policy [3] and Rajkumar's read-write priority ceiling protocol [30]. The latter has also been studied in the context of distributed real-time databases [30].

An alternative to locking is the use of specialized *non-blocking* algorithms [1]. 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]

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

[1]In non-blocking read-write algorithms, the value to be written must be

In this paper, we focus on lock-based RW synchronization in cache-coherent shared-memory multiprocessor systems.

Contributions. The contributions of this paper are as follows: **(i)** We study the applicability of existing RW locks to real-time systems and derive worst-case blocking bounds; **(ii)** we propose a novel type of RW lock based on the concept of "phase-fairness," derive corresponding worst-case blocking bounds, and present two efficient implementations; **(iii)** we 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. These experiments show that employing "phase-fair" RW locks can significantly improve real-time performance.

The rest of this paper is organized as follows. We summarize relevant background in Sec. 2, and discuss our synchronization protocol and other RW lock choices in Sec. 3. An empirical performance evaluation is presented in Sec. 4, followed by our conclusions in Sec. 5.

2 Background

We consider the scheduling of a system of *sporadic tasks*, denoted $T_1, \ldots, T_N$, on m processors. 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*, $\mathsf{r}(T_i^j)$. Each task T_i is specified by its *worst-case (per-job) execution cost*, $\mathsf{e}(T_i)$, its *period*, $\mathsf{p}(T_i)$, and its *relative deadline*, $\mathsf{d}(T_i) \geq \mathsf{e}(T_i)$. A job T_i^j should complete execution by its *absolute deadline*, $\mathsf{r}(T_i^j) + \mathsf{d}(T_i)$, and is *tardy* if it completes later. The spacing between job releases must satisfy $\mathsf{r}(T_i^{j+1}) \geq \mathsf{r}(T_i^j) + \mathsf{p}(T_i)$. A job that has not completed execution is either *preemptable* or *non-preemptable*. A job scheduled on a processor can only be preempted when it is preemptable. Task T_i's *utilization* (or *weight*) reflects the processor share that it requires and is given by $\mathsf{e}(T_i)/\mathsf{p}(T_i)$.

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 [16]). 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 [14, 17].

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 apply 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.

Resources. When a job T_i^j is going to update (observe) the state of a *resource* ℓ, it *issues* a *write* (*read*) *request* $\mathcal{W}_\ell$ ($\mathcal{R}_\ell$) for 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 ℓ of either kind as $\mathcal{X}_\ell$.

$\mathcal{X}_\ell$ is *satisfied* as soon as T_i^j *holds* ℓ, and *completes* when T_i^j *releases* ℓ. $|\mathcal{X}_\ell|$ denotes the maximum time that T_i^j will hold ℓ. T_i^j becomes *blocked* on ℓ if $\mathcal{X}_\ell$ 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 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. A resource is *global* if it can be requested concurrently by jobs scheduled on different processors, and *local* otherwise.

Infrequent requests. Some infrequently read or updated resources may not be requested by every job of a task. We associate a *request period* $\mathsf{rp}(\mathcal{X}) \in \mathbb{N}$ with every request $\mathcal{X}$ to allow infrequent resource requests to be represented in the task model without introducing unnecessary pessimism. The request period limits the maximum request frequency: if jobs of T_i issue $\mathcal{X}$ and $\mathsf{rp}(\mathcal{X}) = k$, then at most every kth job of T_i issues $\mathcal{X}$.

For example, consider a task T_m that reads a sensor ℓ_s at a rate of ten samples per second via a read request $\mathcal{R}_s$ ($\mathsf{p}(T_m) = 100ms$) and that updates a shared variable ℓ_a storing a history of average readings once every two seconds via a write request $\mathcal{W}_a$. In this case, assuming that every job of T_m issues $\mathcal{W}_a$ when analyzing contention for ℓ_a would be unnecessarily pessimistic. To reflect the fact that only every 20th job of T_m accesses ℓ_a, we can define $\mathsf{rp}(\mathcal{W}_a) = 20$ and $\mathsf{rp}(\mathcal{R}_s) = 1$.

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.

3 Reader-Writer Synchronization

The *flexible multiprocessor locking protocol* (FMLP) [8, 12] is a real-time mutex protocol based on the principles of flexibility and simplicity that has been shown to compare favorably to previously-proposed protocols [8, 11]. In this section,

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.

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 via spinning or suspension. 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 short critical sections, since spinning wastes processor time. However, two recent studies have shown that, in terms of schedulability, spinning is usually preferable to suspending when overheads are considered [11, 15]. Based on these trends (and due to space constraints), we restrict our focus to short resources in this paper and delegate RW synchronization of 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 [8].

Resource groups. Nesting, which is required to cause a deadlock, tends to occur somewhat infrequently in practice [10]. 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 $\mathsf{grp}(\ell)$ denote the group that contains 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.[2]

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 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 $\mathcal{X}_\ell$ for a short resource ℓ and $\mathcal{X}_\ell$ is outermost, then T_i^j becomes non-preemptable and executes the corresponding entry protocol for $\mathsf{grp}(\ell)$'s group lock (the details of which are discussed in the next section). $\mathcal{X}_\ell$ is satisfied once T_i^j holds ℓ's group lock. When $\mathcal{X}_\ell$ completes, T_i^j releases the group lock and leaves its non-preemptive section. The execution of an outermost request is illustrated in Fig. 1. If $\mathcal{X}_\ell$ 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 $\mathcal{X}_\ell$ must also be a read request according to the nesting rule. In either case, it is safe to acquire ℓ.

[2]Group-locking is admittedly a *very* simple deadlock avoidance mechanism; however, the FMLP is the first multiprocessor real-time locking protocol that allows nesting of global resources at all. Obtaining a *provably* better protocol remains an interesting open question.

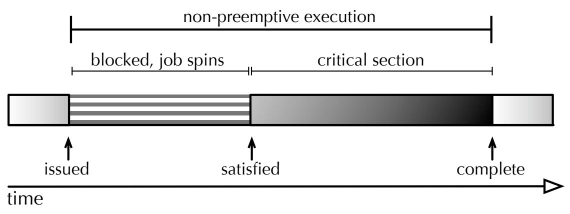

Figure 1: Illustration of an outermost request.

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 showing how using RW locks and relaxing ordering constraints can significantly reduce worst-case blocking. Bounds on worst-case blocking for all three choices are derived in an extended version of this paper [9].

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 Sec. 4) as a performance baseline.

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 [27], and second, they offer strong progress guarantees. Since requests (in the FMLP) are executed non-preemptively, task-fair locks ensure that a request of a job 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 in fact cause deadline misses. An example is shown in Fig. 2(a), which depicts jobs of five tasks (two writers, three readers) competing for a resource ℓ. As ℓ's group lock is a task-fair mutex lock, all requests are satisfied sequentially in the order that they were issued. This unnecessarily delays both T_4 and T_5 and causes them to miss their respective deadlines at times 12.5 and 13.

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 acquired by multiple readers. This can lead to reduced blocking, as shown in Fig. 2(b), which depicts the same arrival sequence as in Fig. 2(a). 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 Fig. 2(c). The only difference in task behavior between Fig. 2(b) and Fig. 2(c) is that the arrival times of the jobs of T_1 and T_4 have been switched. This causes 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.

Preference RW locks. In a reader preference lock, readers are statically prioritized over writers, *i.e.*, writers are starved as long as readers issue consecutive requests [18, 28]. The lack of strong progress guarantees for writers makes reader preference locks a problematic choice for real-time systems: one can easily construct examples in which deadlines are missed due to writer starvation. Writer preference locks are an ill choice for similar reasons.

All previously-proposed RW locks fall within one of the categories discussed so far. Thus, the examples discussed above demonstrate that *no* such lock reliably reduces worst-case blocking (unless there are very few writers).

Phase-fair RW locks. Upon closer inspection, one can identify 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*. Phase-fair RW locks have the following properties: **(i)** *reader phases* and *writer phases* alternate; **(ii)** writers are subject to FIFO ordering with regard to other writers; **(iii)** all unsatisfied read requests are satisfied at the beginning of a reader phase;[3] and **(iv)** if there are incomplete write requests, then read requests are not satisfied until the start of the next reader phase. Properties (i) and (iii) ensure that a read request is never blocked by more than one writer phase and one reader phase *irrespective of m and the length of the write request queue*. Property (iv) ensures that reader phases end. Properties (i) and (ii) ensure that a write request is never blocked by more than $m-1$ phases. The fact that the bound on read request blocking is $O(1)$ and not $O(m)$ is very significant as multicore platforms become larger.

[3]Exactly one write request is satisfied at the beginning of a writer phase.

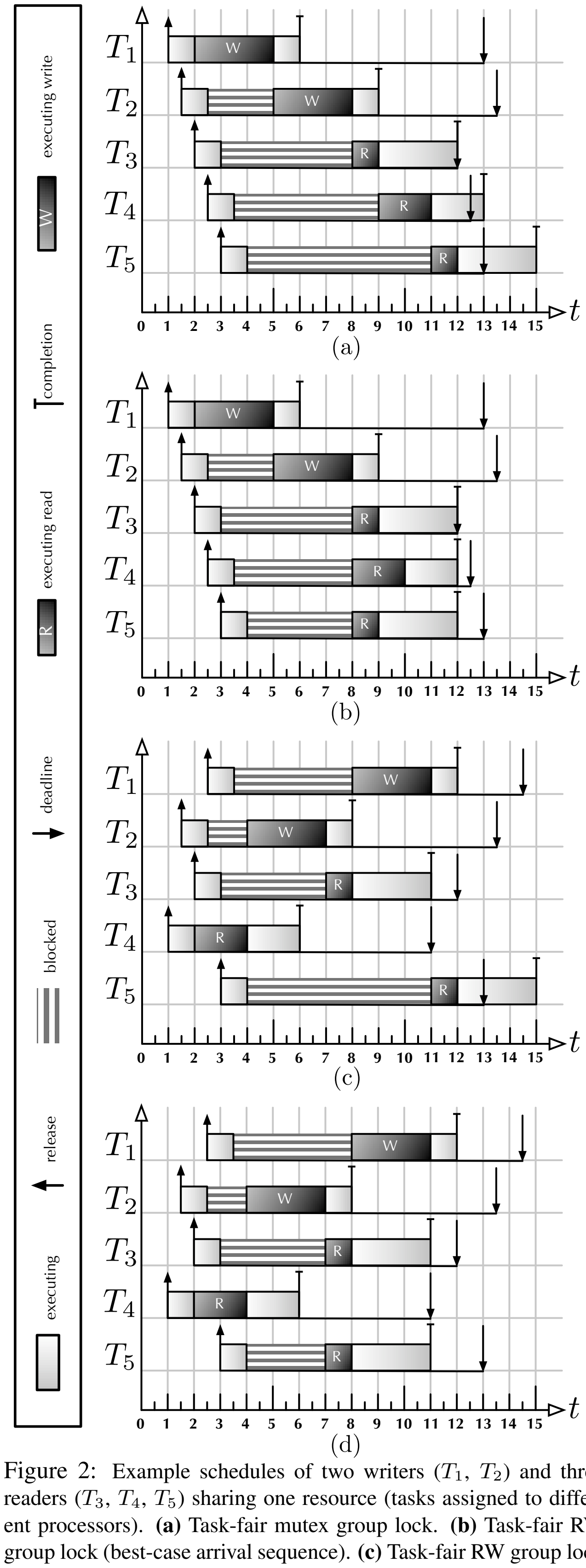

Figure 2: Example schedules of two writers (T_1, T_2) and three readers (T_3, T_4, T_5) sharing one resource (tasks assigned to different processors). **(a)** Task-fair mutex group lock. **(b)** Task-fair RW group lock (best-case arrival sequence). **(c)** Task-fair RW group lock (worst-case arrival sequence). **(d)** Phase-fair RW group lock.

Fig. 2(d) depicts a schedule of the pathological arrival sequence from Fig. 2(c) 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 (iv). 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 (i), this starts the next reader phase, and due to Property (iii), *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 Fig. 2.

3.3 A Phase-Fair Reader-Writer Lock

While the above example suggests that the properties of phase-fairness can reduce worst-case blocking significantly, to be a viable choice, phase-fair locks must be efficiently implementable on common hardware platforms. In this section, we present a simple and efficient phase-fair RW lock that only depends on hardware support for atomic-add, fetch-and-add, and atomic stores. The algorithm, as given in its entirety in Listing 1, assumes a 32-bit little-endian architecture. However, it can be easily adapted to 16-bit, 64-bit, and big-endian architectures.

Structure. The lock consists of four counters that count 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 Fig. 3.

Readers. The reader entry procedure (lines 10–13) 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).

```
type pflock = 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 : ^pflock)                              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 : ^pflock)                            15
  atomic_add(&L->rout, RINC)                                  16

procedure write_lock(L : ^pflock)                             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 : ^pflock)                           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: Implementation of a phase-fair spin-based reader-writer lock (algorithm RW-PF).

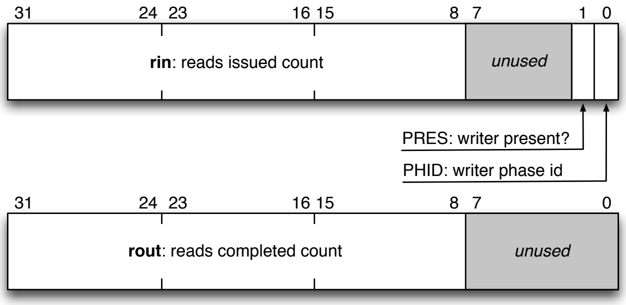

Figure 3: The allocation of bits in the reader entry counter and the reader exit counter (corresponding to Line 2 of Listing 1).

Writers. Similarly to Mellor-Crummey and Scott's simple task-fair RW lock [28], FIFO ordering of writers is realized with a ticket abstraction [27]. 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

Algorithm	Name	Complexity	Fairness
Mutex Ticket Lock [27]	MX-T	$O(m)$	task-fair
Mutex Queue Lock [27]	MX-Q	$O(1)$	task-fair
Reader-Writer Ticket Lock [28]	RW-T	$O(m)$	task-fair
Reader-Writer Queue Lock [28]	RW-Q	$O(1)$	task-fair
Phase-Fair Lock (Listing 1)	RW-PF	$O(m)$	phase-fair

Table 1: Local-spin locking algorithms considered in this paper. The complexity metric is remote memory references on m cache-coherent processors.

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 off 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.

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. Assuming the absence of *PHID*, if $\mathcal{R}_1$ would fail to observe the short window during which *PRES* is cleared, then it would 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.

Size concerns. Overflowing rin, win, $rout$, and $wout$ is harmless as long as there are at most $2^{32} - 1$ concurrent writers and $2^{24} - 1$ concurrent readers because the counters are only tested for equality. Note that there can be most m concurrent readers and writers under the FMLP since requests are executed non-preemptively.

In Listing 1, the four counters are defined as four-byte integers each for clarity and performance reasons. However, requiring 16 bytes per lock may be excessive in the context of some memory-constrained applications (*e.g.*, embedded systems). We provide a phase-fair RW lock that only requires 4 bytes and supports up to 127 concurrent readers and writers each in [9].

Implementation in LITMUS$^{\text{RT}}$. 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 [27, 28], as well as our phase-fair RW lock, in LITMUS$^{\text{RT}}$, UNC's Linux-derived real-time OS [13, 17]. As seen in Table 1, these locks are denoted MX-T, MX-Q, RW-T, RW-Q, and RW-PF 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.

One commonly-used complexity metric for locking algorithms is to count *remote memory references* (RMR) [2], *i.e.*, on a cache-coherent multiprocessor, locks are classified by how many cache invalidations occur per request under maximum contention. In theory, $O(1)$ locks, *e.g.*, MX-Q, RW-Q, should outperform $O(m)$ locks, *e.g.*, MX-T, RW-T, since frequent cache invalidations cause memory bus contention. However, micro benchmarks in which a lock was accessed repeatedly in a tight loop revealed that—on both x86 and SPARC V9 platforms—the MX-Q and RW-Q locks incur significantly higher overheads than the ticket-based MX-T and RW-T locks. The reason is that the $O(1)$ locks inherently require additional slow compare-and-swap instructions. In our micro benchmarks, the positive effects of their lower RMR complexity (*i.e.*, reduced bus traffic) only starts to outweigh the increased entry and exit overheads at unrealistic contention levels, *i.e.*, spinning must comprise a majority of the system's load in order to break even, which is rather unlikely to occur in well-designed real-time systems. Hence, we only consider the MX-T and RW-T locks in the rest of this paper.

4 Experimental 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 MX-T, RW-T, and RW-PF group locks. The methodology followed in this study, and our results, are discussed below.

4.1 Overheads

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$^{\text{RT}}$ on a Sun UltraSPARC T1 "Niagara" multicore platform. The Niagara is a 64-bit machine containing eight cores on one chip running at 1.2 GHz. Each core supports four hardware threads, for a total of 32 logical processors. On-chip caches include a 16K (respective, 8K) four-way set associative L1 instruction (respective, 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. Most relevant system overheads (*e.g.*, scheduling overhead, context-switch overhead, *etc.*) were already known from a recent study [14] and did not have to be re-determined. Only synchronization-related overheads had to be obtained and are given in Table 2. When determining

Overhead	Worst-Case	Average-Case
task-fair mutex: read/write request	0.130	0.129
task-fair reader-writer lock: read request	0.160	0.157
task-fair reader-writer lock: write request	0.154	0.153
task-fair phase-fair lock: read request	0.144	0.142
task-fair phase-fair lock: write request	0.180	0.178
leaving non-preemptive section	2.137	1.570

Table 2: Worst-case and average-case costs of synchronization overheads (in μs). Based on the methodology explained in detail in [14], 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.

HRT (SRT) schedulability, we assume worst-case (average-case) overheads (as in [14]).

4.2 Task Set Generation

In generating random task sets for conduction schedulability comparisons, task parameters were selected—similar to the approach previously used in [11, 14, 15]—as follows. Task utilizations were distributed uniformly over $[0.1, 0.4]$ and periods were chosen from $[10ms, 100ms]$. We only considered implicit deadlines, *i.e.*, $\mathsf{p}(T_i) = \mathsf{d}(T_i)$, since most G-EDF schedulability tests require this constraint. 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. We selected these parameter ranges because they correspond to the "medium weight distribution" previously considered in [14]. The effect of choosing heavier or lighter utilization distributions is considered in the discussion of the results below.

Resource sharing. 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 equalled Q_r (Q_w). Request density is the normalized rate at which a request is issued.[4] We chose a request period of one unless that would have exceeded Q_r (Q_w).[5] Based on the *nesting probability* ($nest$), 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, some tasks may not request resources at all, and 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 [11, 15], which in turn were based on durations observed in actual systems [10]. The effect of allowing longer durations is discussed below.

[4]A request $\mathcal{X}$ assigned to T_i has a density of $\frac{1000ms}{\mathsf{p}(T_i)\cdot \mathsf{rp}(\mathcal{X})}$. The factor of 1000ms is due to the use of one second as the normalization interval.

[5]Larger request periods are common for low $wratios$.

4.3 Schedulability Tests

After a task system was generated, its schedulability assuming MX-T, RW-T, and RW-PF group locks was tested as follows. System and synchronization overheads were accounted for by inflating worst-case execution costs and the durations of outermost requests using standard techniques [26]. Per-task bounds on worst-case blocking were computed as detailed in [9], 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 $m = 32$ [19]. Determining whether a task system is hard-schedulable under G-EDF is more involved. There now are five major sufficient (but not necessary) HRT schedulability tests for G-EDF [4, 5, 6, 7, 20]. Interestingly, for each of these tests, there exist task sets that are deemed schedulable by it but not the others [5, 7]. 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 (HRT and SRT schedulablity under P-EDF is the same except for the use of worst-case versus average-case overheads.) [25]. Note that partitioning must precede the blocking-term calculation.

4.4 Study

In our study, we assessed SRT/HRT schedulability under both G-EDF and P-EDF while varying five parameters: **(i)** $ucap \in [1.0, 32.0]$, **(ii)** $contention \in [50, 550]$, **(iii)** $wratio \in [0.01, 0.5]$, **(iv)** $nest \in [0.0, 0.5]$, and **(v)** $res \in [0.1, 5.0]$. Each of these parameters was varied over its stated range for *all* possible choices of the other parameters arising from $ucap \in \{6.0, 9.0, 12.0\}$ for HRT and $ucap \in \{15.0, 18.0, 21.0\}$ for SRT, $contention \in \{100, 250, 400\}$, $wratio \in \{0.05, 0.2, 0.35\}$, $nest \in \{0.05, 0.2, 0.35\}$, and $res \in \{0.5, 2.0, 3.5\}$, under both SRT and HRT and both G-EDF and P-EDF scheduling. 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 1,620 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 these observations.

In the majority of the tested scenarios, RW-PF locks were clearly the best-performing algorithm. MX-T locks were preferable in only 23 of the 1,620 tested scenarios (12 under G-EDF, 11 under P-EDF). 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 $ucap$, $contention$, $wratio$, $nest$)—either (almost) all or none of the task sets were schedulable in these scenarios regardless of the group lock type. Where RW synchronization was preferable to mutual exclusion, RW-T locks *never* outperformed RW-PF locks.

Generally speaking, RW-PF 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 RW-PF locks (and why), and **(ii)** under which conditions do RW-T locks perform as well as RW-PF locks? Regarding (i), all 23 cases exhibit a combination of high request density, deep and frequent nesting, and many writers ($wratio \geq 0.35$). Hence, writer blocking, which is not improved by RW locks, becomes the dominating performance factor. This explains why RW-PF (and RW-T) locks do not perform better than MX-T locks in some cases, but why do they perform *worse*? The reason is additional pessimism in the worst-case blocking analysis of RW-PF locks that is only triggered by scenarios that involve many writers and frequent, long requests (see [9]). The answer to (ii) reinforces the intuition that task-fair locks are very sensitive to the number of concurrent writers: *all* scenarios in which RW-T locks perform as well as RW-PF 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 RW-PF locks clearly perform better than RW-T locks (in terms of schedulability) where $wratio = 0.05$.

Example graphs. Insets (a)-(j) of Fig. 4 display ten selected graphs that illustrate the above trends. In order to show a wide variety of scenarios, we chose to exhibit one HRT schedulability result under P-EDF (left column) and one SRT schedulability result under G-EDF (right column) for each of the five parameters that we varied (rows).

Insets (a)-(b) show schedulability as a function of $ucap$. In both cases, RW-PF locks yield better schedulability: under P-EDF (resp., G-EDF), with MX-T and RW-T locks, performance starts to degrade $ucap \approx 8$ (resp., $ucap \approx 13$), whereas RW-PF locks can sustain high schedulability until $ucap \approx 10$ (resp., $ucap \approx 16$). Note that with only 20% writes RW-T locks perform almost as badly as MX-T locks. Insets (c)-(d) show schedulability as a function of $contention$. For both P-EDF and G-EDF, RW-PF locks can sustain almost twice as much contention as either MX-T or RW-T locks before performance degrades. This highlights the significance of the RW-PF lock $O(1)$ bound on read blocking under high contention. Again, RW-T locks offer little advantage over MX-T locks. Insets (e)-(f) show schedulability as a function of $wratio$. Inset (e) illustrates under which conditions MX-T locks are (partially) preferable. Due to high $contention$, $nest$, and res, RW-PF locks start to perform worse than MX-T locks when $wratio \geq 0.35$ due to pessimism in the analysis with regard to high number of writers. In contrast, inset (f) shows a more representative scenario in which RW-PF locks can sustain high schedulability until $wratio \approx 0.3$, whereas RW-T locks are only usable when there are virtually no writes even with moderate $nest$ and res. Insets (g)-(h) show schedulability as a function of $nest$. In both cases, RW-PF locks perform significantly better than either MX-T and RW-T locks as resource groups become fewer in number and larger, even under high res (P-EDF case) and high $contention$ (G-EDF case). Finally, insets (i)-(j) show schedulability as a function of res. Once again, RW-PF locks can sustain significantly higher schedulability and RW-T locks perform only little better than MX-T locks.

Varying weights, periods, and request lengths. With lighter utilization distributions, the differences between the locks would be magnified since the number of tasks increases and bounds on worst-case blocking become larger relative to the average worst-case execution cost. Similarly, longer request durations would also magnify differences in performance. Lengthening periods causes a decrease in average per-request density, which in turn causes the number of requests per task to increase. Further, the interval for which interference needs to be analyzed increases [9]. In both cases, the bounds on worst-case blocking would be more pessimistic, and hence differences in performance would be magnified.

5 Conclusion

We have presented the first analysis of reader-writer locking in shared-memory multiprocessor real-time systems and demonstrated that preference- and task-fair locks can be problematic in such systems. We also have proposed phase-fair locks, an alternative lock design with asymptotically lower worst-case read blocking. Our experiments revealed that (in terms of schedulability) phase-fair locks are almost always the best choice—oftentimes by a significant margin.

In future work, we would like to extend the RW-FMLP

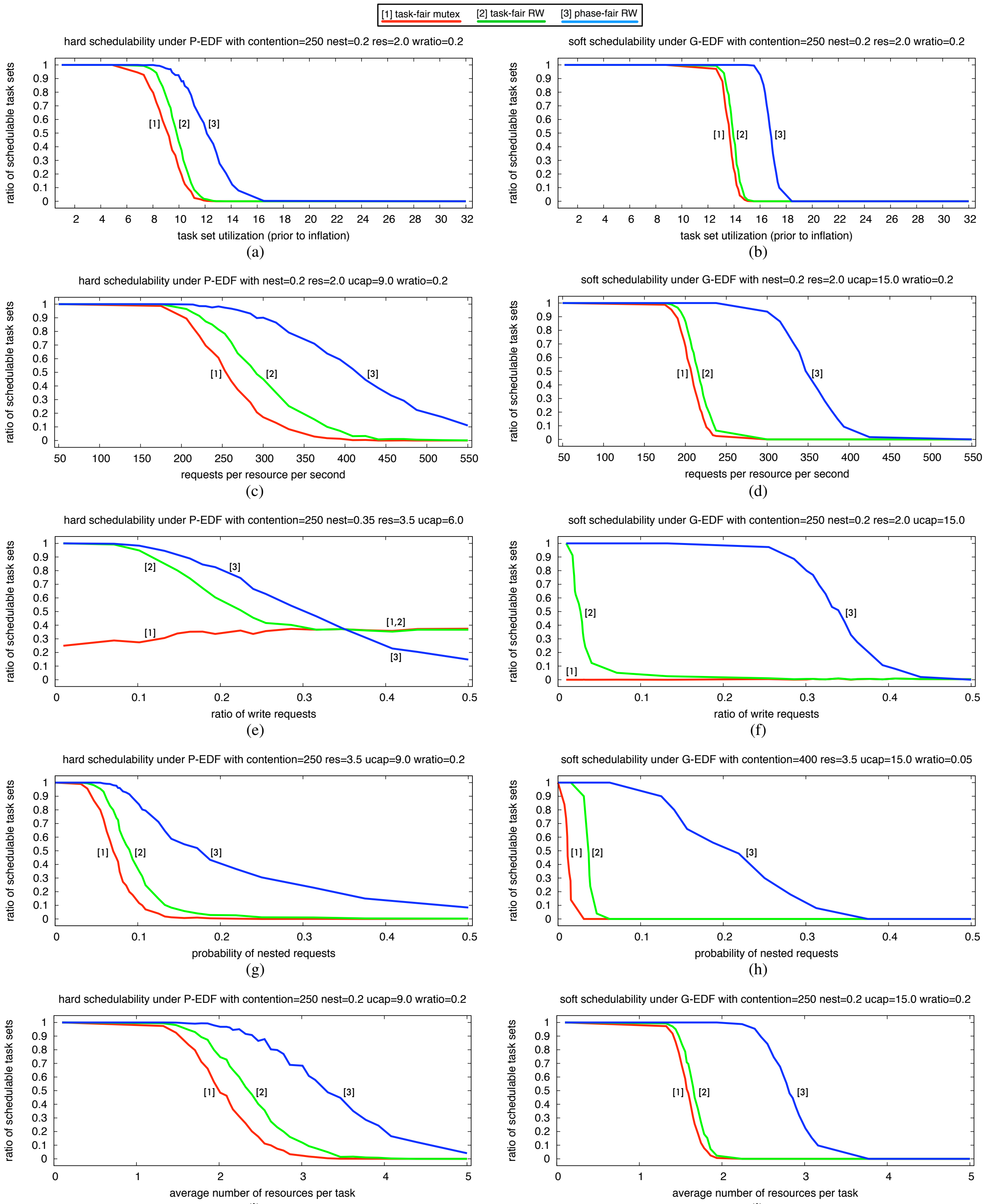

Figure 4: Schedulability (the fraction of generated task systems deemed schedulable) as a function of **(a)-(b)** task system utilization ($ucap$), **(c)-(d)** average requests per resource per second ($contention$), **(e)-(f)** ratio of write requests ($wratio$), **(g)-(h)** probability of nested requests ($nest$), **(i)-(j)** average number of resources per task (res). The left column shows hard schedulability under P-EDF; the right right column shows soft schedulability under G-EDF. The y-axis of each graph gives the fraction of successfully-scheduled task sets. The scenario considered in each graph is indicated above that graph.

to support suspension-based real-time reader-writer synchronization. Further, we would like to develop a phase-fair reader-writer lock with $O(1)$ RMR complexity and re-evaluate the performance of reader-writer lock choices in the absence of shared, coherent caches.

References

[1] J. Anderson and P. Holman. Efficient pure-buffer algorithms for real-time systems. In *Proc. of the Seventh International Conference on Real-Time Systems and Applications*, pp. 57–64, 2000.

[2] J. Anderson, Y. Kim, and T. Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing*, 16(2-3):75–110, 2003.

[3] T. Baker. Stack-based scheduling for realtime processes. *Real-Time Systems*, 3(1):67–99, 1991.

[4] T. Baker. Multiprocessor EDF and deadline monotonic schedulability analysis. In *Proc. of the 24th IEEE Real-Time Systems Symposium*, pp. 120–129, 2003.

[5] Sanjoy Baruah. Techniques for multiprocessor global schedulability analysis. In *Proc. of the 28th IEEE International Real-Time Systems Symposium*, pp. 119–128, 2007.

[6] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Improved schedulability analysis of edf on multiprocessor platforms. In *Proc. of the 17th Euromicro Conference on Real-Time Systems*, pp. 209–218, 2005.

[7] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on Parallel and Distributed Systems*, 99(1):(to appear), 2008.

[8] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proc. of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 47–57, 2007.

[9] B. Brandenburg and J. Anderson. Reader-writer synchronization for shared-memory multiprocessor real-time systems (extended version), 2009.
Available at http://www.cs.unc.edu/~anderson/papers.html.

[10] B. Brandenburg and J. Anderson. Feather-trace: A light-weight event tracing toolkit. In *Proc. of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pp. 20–27, 2007.

[11] B. Brandenburg and J. Anderson. A comparison of the M-PCP, D-PCP, and FMLP on LITMUS$^{\mathrm{RT}}$. In *Proc. of the 12th International Conference On Principles Of Distributed Systems*, pp. 105–124, 2008.

[12] B. Brandenburg and J. Anderson. An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in LITMUS$^{\mathrm{RT}}$. In *Proc. of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 185–194, 2008.

[13] B. Brandenburg, A. Block, J. Calandrino, U. Devi, H. Leontyev, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A status report. In *Proc. of the 9th Real-Time Linux Workshop*, pp. 107–123, 2007.

[14] B. Brandenburg, J. Calandrino, and J. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Proc. of the 29th IEEE Real-Time Systems Symposium*, pp. 157–169, 2008.

[15] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Synchronization on real-time multiprocessors: To block or not to block, to suspend or spin? In *Proc. of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 342–353, 2008.

[16] J. Calandrino, J. Anderson, and D. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proc. of the 19th Euromicro Conference on Real-Time Systems*, pp. 247–256, 2007.

[17] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proc. of the 27th IEEE Real-Time Systems Symposium*, pp. 111–123, 2006.

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

[19] U. Devi and J. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. *Real-Time Systems*, 38(2):133–189, 2008.

[20] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2-3):187–205, 2003.

[21] P. Gore, I. Pyarali, C. Gill, and D. Schmidt. The design and performance of a real-time notification service. In *Proc. of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 112–120, 2004.

[22] W. Hsieh and W. Weihl. Scalable reader-writer locks for parallel systems. In *Proc. of the 6th International Parallel Processing Symposium*, pp. 656–659, 1992.

[23] O. Krieger, M. Stumm, R. Unrau, and J. Hanna. A fair fast scalable reader-writer lock. In *Proc. of the 1993 International Conference on Parallel Processing*, pp. 201–204, 1993.

[24] Q. Li and C. Yao. *Real-Time Concepts for Embedded Systems*. CMP Books, 2003.

[25] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 30:46–61, 1973.

[26] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[27] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[28] J. Mellor-Crummey and M. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proc. of the 3rd ACM SIGPLAN symposium on principles and practice of parallel programming*, pp. 106–113, 1991.

[29] M. Musial, V. Remuß, C. Deeg, and G. Hommel. Embedded system architecture of the second generation autonomous unmanned aerial vehicle MARVIN MARK II. In *Proc. of the 7th International Workshop on Embedded Systems-Modeling, Technology and Applications*, pp. 101–110, 2006.

[30] R. Rajkumar. *Synchronization In Real-Time Systems – A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[31] M. Reiman and P. Wright. Performance analysis of concurrent-read exclusive-write. In *Proc. of the 1991 ACM SIGMETRICS Conference on Measurement and modeling of computer systems*, pp. 168–177, 1991.