Locking in Pfair-scheduled Multiprocessor Systems*

Philip Holman and James H. Anderson

Department of Computer Science

University of North Carolina

Chapel Hill, NC 27599-3175

Phone: (919) 962-1757

Fax: (919) 962-1799

E-mail: {holman, anderson}@cs.unc.edu

May 2002

Abstract

We present several locking synchronization protocols and associated schedulability conditions for Pfair-scheduled multiprocessor systems. We focus on two classes of protocols. The first class is only applicable in systems in which all critical sections are short relative to the length of the scheduling quantum. In this case, efficient synchronization can be achieved by ensuring that all locks have been released before tasks are preempted. The second and more general protocol class is applicable to any system. For this class, we propose the use of statically-weighted resource servers. We also discuss several inheritance-based protocols as possible alternatives.

*Work supported by NSF grants CCR 9972211, CCR 9988327, and ITR 0082866.

1 Introduction

In recent years, there has been considerable interest in fair scheduling algorithms for multiprocessor systems [3, 4, 5, 7, 8, 9, 16, 23]. Under fair disciplines, each task is assigned a weight that represents its share of the system's resources. At present, fair scheduling algorithms are the only known means for optimally scheduling recurrent real-time tasks on multiprocessors, and thus are of importance from a theoretical perspective. In addition, there has been growing practical interest in such algorithms. Ensim Corp., for example, an Internet service provider, has deployed multiprocessor fair scheduling algorithms in its product line [9].

One limitation of most prior work on multiprocessor fair scheduling algorithms is that only *independent* tasks that do not synchronize or share resources have been considered. In contrast, tasks in real systems usually are not independent. Synchronization entails additional overhead, which must be taken into account when determining system feasibility [2, 6, 18, 19, 20, 22]. Unfortunately, prior work on real-time synchronization has been directed at uniprocessor systems, or systems implemented using non-fair scheduling algorithms (or both), and thus cannot be directly applied in fair-scheduled multiprocessor systems. (Indeed, synchronization issues in fair-scheduled *uniprocessor* systems and related bandwidth-preserving server schemes were first considered only very recently [10, 11, 14, 16].)

In recent work [13], we demonstrated the effectiveness of lock-free algorithms for implementing shared objects in fair-scheduled systems. Lock-free algorithms do not use semaphores to synchronize tasks. Hence, tasks never block. Blocking can be particularly problematic in fair-scheduled systems because blocking can disrupt fair allocation rates.

Although lock-free algorithms are ideal for fair-scheduled systems, they are not always appropriate. Specifically, lock-free techniques tend to produce efficient implementations only for simple objects, such as queues and buffers, and often require strong synchronization primitives. For complex objects, overhead (both time *and* space) is often prohibitive. In such cases, semaphore-based locking techniques are more suitable. Furthermore, lock-based synchronization may also be needed to synchronize accesses to external devices.

In this paper, we consider the problem of incorporating lock-based synchronization into fair-scheduled multiprocessor systems. As in our previous work, the notion of fairness that we consider is the *Pfairness* constraint proposed by Baruah *et al.* [7]. We also limit attention to *periodic* task systems [15]. Although we consider only Pfair-scheduled periodic tasks, many of our results are applicable to other fair scheduling algorithms and notions of recurrent execution as well.

We present several locking protocols that are optimized for the special case in which critical sections require only a small fraction of a scheduling quantum. Although we expect these protocols to be widely applicable, we also present a simple server-based protocol that permits arbitrarily long critical sections. After describing and analyzing these protocols, we present an experimental evaluation that demonstrates both the relative performance of each in various situations and the inherent cost of using long critical sections in Pfair-scheduled systems.

The rest of this paper is organized as follows. In Sec. 2, we present background information and state assumptions relating to our work. We then present our optimized protocols in Sec. 3, followed by the server-based protocol in Sec. 4. Experimental results are presented in Sec. 5. We conclude in Sec. 6.

2 Background

Let τ denote a set of N periodic tasks[1] to be scheduled on $M \geq 1$ processors. Let ρ denote the set of locks shared by tasks in τ. Assume that each lock $\ell \in \rho$ is requested by at least two tasks and that each task requests at least one lock. (This last assumption simplifies the presentation of our results. The system may also contain other tasks that do not require synchronization. However, due to fair scheduling, these tasks are isolated from tasks in τ and, hence, neither affect nor are affected by synchronization overheads.)

Task model. Under Pfair scheduling, each task $T \in \tau$ has a *weight* $T.w$ in the range (0,1]. Conceptually, the weight of a task T is the fraction of a single processor to which T is entitled. $T.w$ is determined (as explained later) by considering two parameters associated with T: an integer *period* $T.p$, and an *execution cost* $T.e$, which is defined as the worst-case cost of a single job (*i.e.*, invocation) of T. Unlike prior work on Pfair scheduling, we do *not* assume that $T.e$ is an integer.

We further decompose each task T into a sequence of $T.n$ phases, $T^{[1]} \ldots T^{[T.n]}$. In each phase $T^{[i]}$, T requires *at most one* lock,[2] as specified by $L(T^{[i]}) \in \rho \cup \{\emptyset\}$, and requires at most $T^{[i]}.e$ units of execution time. $L(T^{[i]}) = \emptyset$ implies that phase $T^{[i]}$ requires no locks. If $L(T^{[i]}) \neq \emptyset$, then $T^{[i]}$ is a *locking phase* or *critical section* of T. Otherwise, $T^{[i]}$ is a *non-locking phase* of T. We assume that $L(T^{[i]}) \neq L(T^{[i+1]})$ and that $L(T^{[i]}) \neq \emptyset \Leftrightarrow L(T^{[i+1]}) = \emptyset$ for all $i < T.n$. (This assumption is only made to simplify the presentation of our results.) $T.e$, mentioned previously, is defined as $T.e = \sum_{i=1}^{T.n} T^{[i]}.e$.

Pfair scheduling. Pfair scheduling algorithms allocate processor time in discrete time units, or quanta. The interval $[t, t+1)$, where t is a non-negative integer, is called *time slot* t. In each time slot, each processor can be assigned to at most one task and each task can be assigned at most one processor. Task migration is allowed. The sequence of scheduling decisions made in these time slots defines a *schedule*. A schedule respects Pfairness if and only if the following property holds for all $T \in \tau$ and for all $t \geq 0$ [12].

Pfairness (Pf): T receives either $\lfloor T.w \cdot t \rfloor$ or $\lceil T.w \cdot t \rceil$ quanta over any interval $[0, t)$.

Baruah *et al.* [7] showed that a schedule satisfying Property (Pf) exists for a task set τ on M processors if and only if the following condition holds.

$$\sum_{T \in \tau} T.w \leq M \tag{1}$$

[1]We assume each task begins execution at time 0.

[2]Note that this model does not allow nested critical sections, and thus deadlock cannot occur. Since our focus in this paper is to introduce the concept of locking in a fair-scheduled environment, we have intentionally chosen a simplified model. Issues such as nested accesses and deadlock avoidance are left as future work

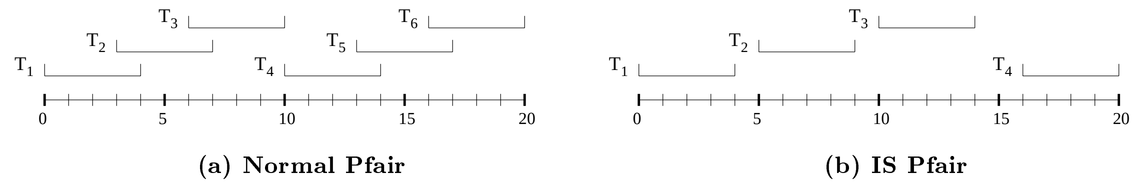


Figure 1: Windowing for a task with weight $T.w = 3/10$ when **(a)** all windows are released on time, and **(b)** the release of each window after the first introduces an intra-sporadic delay of two slots.

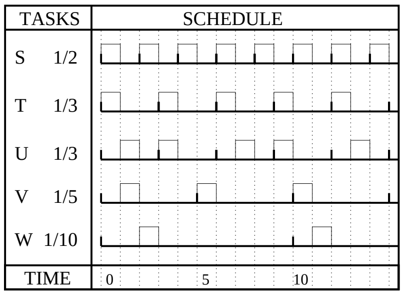
Property (Pf) effectively sub-divides each task T into a series of evenly-distributed quantum-length *subtasks*. Adopting the notation of Anderson and Srinivasan [3], we let T_i denote the ith subtask of T. For example, Fig. 1(a) shows the time slot interval within which each subtask must execute to satisfy Property (Pf) when $T.w = 3/10$. The interval associated with each subtask T_i is called T_i's *window* and is denoted $\omega(T_i)$. For example, $\omega(T_2)$ in Fig. 1(a) is $[3, 6]$. (Recall that the ith time slot spans the time interval $[i, i+1)$. Therefore, the time-slot interval $[3, 6]$ corresponds to the time interval $[3, 7)$.) $\omega(T_i)$ extends from the *pseudo-release* of T_i, denoted $r(T_i)$, to its *pseudo-deadline*, denoted $d(T_i)$, both of which are measured in time slots. In Fig. 1(a), $r(T_2) = 3$ and $d(T_2) = 6$. The "pseudo" prefix avoids confusion with job releases and deadlines and may be omitted when the proper interpretation is clearly implied.

At present, three optimal Pfair scheduling algorithms are known: PF [7], PD[8], and PD^2 [3, 5, 23]. Each algorithm prioritizes subtasks using an *earliest-pseudo-deadline-first* (EPDF) rule, in which priority is given to the eligible subtask with the earliest deadline. However, these algorithms differ in their choice of tie-breaking rules that are used when two subtasks have matching deadlines. (See [3, 5, 7, 8, 23] for details.) We consider only PD^2 in the remainder of the paper since it is the most efficient of these algorithms.

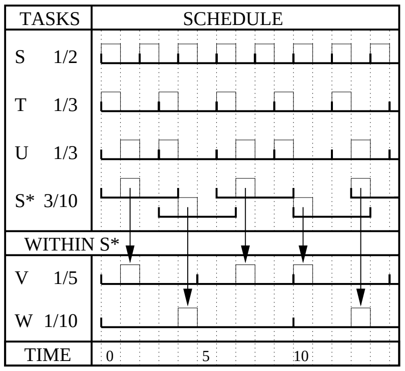
Fig. 2(a) shows a PD^2 schedule in which two processors are shared among five tasks, labeled $S, \ldots, W$, that are assigned weights 1/2, 1/3, 1/3, 1/5, and 1/10, respectively. Vertical dashed lines mark the time-slot boundaries and boxes show when each task is scheduled.

The intra-sporadic task model. Srinivasan and Anderson [23] proved that PD^2 also schedules any *intra-sporadic* (IS) task set that satisfies (1). Under this model, a subtask release can be delayed as long as the relative separation between each pair of windows is at least as long as it would have been had no delay occurred. For example, suppose that T_2's release is delayed for two slots in Fig. 1(a), *i.e.*, it is released at time 5 instead of time 3. Releasing T_3 at time 6 (as before) results in a relative separation of only one between $r(T_2)$ and $r(T_3)$. Since the relative separation is three without the delay (see Fig. 1(a)), the separation restriction is violated. To avoid this violation, T_3's release must be delayed until at least slot 8. Fig. 1(b) illustrates one acceptable window layout for an IS task with weight 3/10.

Scheduling periodic tasks using Pfairness. All previous work on Pfair-scheduled periodic task systems hinges on the following property, first observed by Baruah *et al.* [7], which follows from Property (Pf).



(a) PD^2 schedule



(b) Supertasking PD^2 schedule

Figure 2: Sample Pfair schedules for a task set consisting of five tasks with weights 1/2, 1/3, 1/3, 1/5, and 1/10, respectively. **(a)** Normal schedule produced when no supertasks are used. **(b)** Schedule produced when tasks V and W are combined into the supertask S*, which competes with weight 3/10.

Periodicity (P): If $T.w = b/c$ for integers $c \geq b \geq 1$, then T is scheduled in *exactly* b time slots within $[k \cdot c, (k+1) \cdot c - 1]$ for all integers $k \geq 0$ in *any* schedule that respects Pfairness.

Property (Pf) implies that windows $\omega(T_{k \cdot b})$ and $\omega(T_{k \cdot b+1})$ are disjoint for all $k \geq 1$.[3] Hence, exactly b windows lie within the time interval $[k \cdot c, (k+1) \cdot c)$ for all $k \geq 0$. Fig. 1(a) shows the first two such intervals (corresponding to $k = 0$ and $k = 1$) for a task of weight 3/10. According to (P), exactly three windows lie within $[10k, 10(k+1))$ for all $k \geq 0$ when $b = 3$ and $c = 10$.

Supertasking. Under *supertasking* [17, 12], the task set τ is partitioned into a collection of non-empty subsets, σ. Each $\mathcal{S} \in \sigma$, called a *supertask*, is assigned a weight $\mathcal{S}.w$ (see below) in the range (0,1] and is then scheduled in place of the tasks in $\mathcal{S}$, called $\mathcal{S}$'s *component tasks*. We let $\mathcal{S}(T)$ denote the supertask of which task T is a component. Whenever a supertask is scheduled, it selects one of its component tasks to receive the quantum. (If there is only one component task, then it may be scheduled directly.) Effectively, a supertask behaves like a *virtual uniprocessor* and may be used to restrict which tasks can execute in parallel.

The schedule shown in Fig. 2(b) is derived from that of Fig. 2(a) by letting $\sigma = \{\{S\}, \{T\}, \{U\}, \{V, W\}\}$. Here, S^* is a supertask with component tasks V and W that competes with weight $S^*.w = 3/10 = V.w + W.w$. The arrows show S^* passing its allocation (upper schedule) to one of its component tasks (lower schedule).

Although this example uses a supertask weight equal to the cumulative weight of the component tasks, Moir and Ramamurthy [17] demonstrated that such a weight assignment may result in deadline misses when used with PF, PD, or PD^2. In previous work, we demonstrated that supertasks can guarantee the safety of their component tasks by using a slightly larger weight [12]. In the remainder of the paper, we refer to the process of selecting such an inflated supertask weight as *reweighting*. It remains an open problem whether supertasking without reweighting is inherently suboptimal.

[3] Under Pfair scheduling, $\omega(T_i)$ can overlap with $\omega(T_{i+1})$ by at most one time slot. An overlap occurs when $d(T_i) = r(T_{i+1})$.

Locking. We assume that each lock $\ell \in \rho$ is requested by a call to `lock(`ℓ`)` and released by a call to `unlock(`ℓ`)`. If $T^{[i]}$ is a locking phase, then we include the overhead of both calls in $T^{[i]}.e$. Each lock ℓ has a FIFO-ordered wait queue, $\mathsf{WAIT}(\ell)$, in which each lock-requesting task is stored (in a suspended state) until granted the lock. We consider FIFO queuing primarily because the prioritizations used by Pfair schedulers may not be not constant across the duration of a critical section. In addition, the use of FIFO queuing avoids starvation. We say that a *lock-requesting* task T *issues* a request for ℓ by invoking `lock(`ℓ`)`. Once ℓ is granted to T, we refer to T as the *lock-holding* task until it completes the corresponding call to `unlock(`ℓ`)`. Finally, we say that task U has a *competing request* for ℓ with respect to T (and, hence, U is a *competitor* of T for ℓ) if both U and T have issued a request for ℓ.

3 Short Critical Section Protocols

In this section, we present a simple method for implementing locks when all critical sections guarded by a common lock are short relative to the length of the scheduling quantum. In experiments conducted by Ramamurthy [21] on a 66 MHz processor, critical-section durations for a variety of common objects (*e.g.*, queues, linked lists, *etc.*) were found to be in the range of tens of microseconds. On modern processors, these operations will likely require no more than a few microseconds. Since quantum sizes typically range from hundreds of microseconds to milliseconds, we expect the protocols described below to be widely applicable. However, some critical sections, such as those that communicate with external hardware, may be too long to use the protocols proposed in this section. In Sec. 4, we consider alternative techniques for such cases.

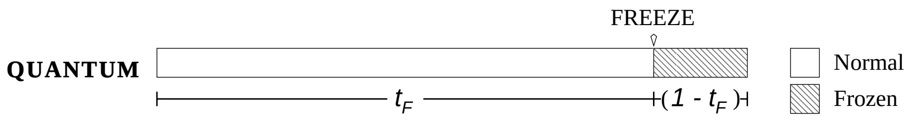
Concept. When some task T is granted a lock, each task U requesting the same lock necessarily experiences a delay of at least the length of T's critical section. However, this unavoidable delay may be amplified drastically if T is preempted. Any such preemption effectively *lengthens* T's critical section. In fair-scheduled systems, this is particularly problematic because each task's allocation is required to be evenly distributed. Hence, if T has a low weight, then any preemption of it will likely result in a long delay.

Fortunately, when using quantum-based scheduling with sufficiently short critical sections, the preemption of lock-holding tasks is avoidable. In particular, when T's critical-section duration is shorter than the scheduling quantum, its execution may span at most two quanta, which implies that T can be preempted at most once before relinquishing the lock. On the other hand, if T's critical section always begins sufficiently early within a quantum, then T will always release the lock before the next quantum boundary. Hence, T will never be preempted while holding a lock.

Because Pfair-scheduled systems are tightly synchronized, *i.e.*, time slots are assumed to be aligned on all processors, we can ensure that each critical-section execution begins sufficiently early in a quantum by using a timer[4] for each lock ℓ that, when triggered, prevents any task from obtaining ℓ for the remainder of

[4]We assume that each lock uses a unique timer for simplicity. Our results can also be applied to cases in which a common

the quantum. Effectively, blocking due to preemptions is replaced by a less expensive form of blocking. We use FREEZE(ℓ) to refer to the disabling signal that is generated when lock ℓ's timer is triggered. We refer to the interval of time between the reception of the FREEZE(ℓ) signal and the end of the quantum as the *frozen interval for* ℓ, as illustrated below. (ℓ is omitted from the figure since it is implied. t_F abbreviates $t_F(\ell)$, which is formally defined below.)



Requirements and additional notation. For this approach is to be effective, the following (obvious) requirements must be satisfied.

(R1) A requesting task must eventually be granted the lock.

(R2) Any critical section that executes within a quantum must complete before the end of that quantum.

(R2) requires that the frozen interval must be at least as long as the longest critical section guarded by the lock. Since most critical sections are expected to require only a small fraction of a quantum, this requirement should be easy to satisfy in most cases.

We will now present two protocols based on these timer signals. These protocols differ in their handling of tasks in $\mathsf{WAIT}(\ell)$ during frozen intervals. The first protocol leaves pending requests in $\mathsf{WAIT}(\ell)$ but ignores them until the requesting tasks are scheduled again, while the second protocol discards any pending request, causing the requesting task to re-issue it later. We present both protocols because the implementation overhead, which is system-dependent, will likely determine which protocol performs best.

We now define a few shorthand notations that are used in the analysis later. (A summary of all notational conventions can be found in Appendix A.) First, let $\mathsf{maxsum}_k\{a_1, \ldots, a_n\}$ be the maximum value produced by summing any $\mathsf{min}(k,n)$ elements from the multiset $\{a_1, \ldots, a_n\}$. For example, $\mathsf{maxsum}_2\{2,2,1\} = \mathsf{max}\{2+2, 2+1, 2+1\} = 4$, $\mathsf{maxsum}_4\{2,2,1\} = \mathsf{max}\{2+2+1\} = 5$, and $\mathsf{maxsum}_0\{2,2,1\} = \mathsf{max}\{0\} = 0$.

In addition, let $T.C(\ell) = \mathsf{max}\{T^{[i]}.e \mid L(T^{[i]}) = \ell\}$ denote the worst-case duration of any phase of T requiring lock ℓ. Second, let $T.Q^m(\ell) = \mathsf{maxsum}_{m(M-1)}\{U.C(\ell) \mid U \neq T\}$ (respectively, $T.I(\ell) = \mathsf{sum}\{U.C(\ell) \mid U \neq T\}$) denote the worst-case execution cost of all competitors of T requesting lock ℓ across m quanta (respectively, across any number of quanta). (The maxsum subscript follows from the fact that at most $M-1$ competing requests can be issued in each quantum. Note that $\lim_{m\to\infty} T.Q^m(\ell) = T.I(\ell)$.) Finally, let $t_F(\ell)$ denote the offset, relative to the start of a quantum, at which the FREEZE(ℓ) signal is generated. That is, if a quantum begins at time t, then the FREEZE(ℓ) signal will be generated at time $t + t_F(\ell)$.

In the analysis that follows, a lock-requesting task T may be blocked due to two sources. First, T is *actively blocked* when one of its competitors is granted the lock before T and executes its critical section in

timer is used for a group of locks.

a time slot in which T is scheduled. We will use the term $B(T^{[i]})$ to denote an upper bound on the active blocking experienced by $T^{[i]}$ across its duration. Since a lock cannot be granted during a frozen interval, T's request may be delayed even if no competing requests are present. We refer to delays caused by the frozen intervals as *freeze blocking*. The duration of freeze blocking is upper bounded by the expression $k(T^{[i]}) \cdot (1 - t_F(L(T^{[i]})))$, where $k(T^{[i]})$ is the number of frozen intervals crossed in the worst case. Given these quantities, phase $T^{[i]}$ requires at most $T^{[i]}.e + B(T^{[i]}) + k(T^{[i]}) \cdot (1 - t_F(L(T^{[i]})))$ quanta. (Assume that $k(T^{[i]}) = 0$ and $B(T^{[i]}) = 0$ for the trivial case in which $L(T^{[i]}) = \emptyset$.) By Property (P), the following weight assignment is therefore sufficient to ensure that all job deadlines are met.

$$T.w = \frac{\left\lceil T.e + \sum_{i=1}^{T.n} (B(T^{[i]}) + k(T^{[i]}) \cdot (1 - t_F(L(T^{[i]})))\right\rceil}{T.p} \tag{2}$$

As an example, consider a task T with three phases and $T.p = 200$. Suppose that $T^{[1]}.e = 1.5$, $L(T^{[1]}) = \emptyset$, $T^{[2]}.e = 0.15$, $L(T^{[2]}) = \mathcal{L}$, $T^{[3]}.e = 4.2$, $L(T^{[3]}) = \emptyset$, and $t_F(\mathcal{L}) = 0.95$. Furthermore, suppose that an analysis of the system determines that $B(T^{[2]}) = 0.35$ and $k(T^{[2]}) = 2$. It follows that an upper bound on the execution requirement of $T^{[2]}$ is $T^{[2]}.e + B(T^{[2]}) + k(T^{[2]}) \cdot (1 - t_F(L(T^{[2]}))) = 0.15 + 0.35 + 2 \cdot (1 - 0.95) = 0.6$. Since an upper bound on the execution requirement of each phase is now known, we can apply (2) to produce the weight assignment of $\lceil 1.5 + 0.6 + 4.2 \rceil / 200 = 7/200$.

The protocols presented in this paper have the advantage that they can be used together in the same system. For example, suppose that we add another locking phase $T^{[4]}$ to T, but that the timer-based protocols considered here cannot be used to implement lock $L(T^{[4]})$ due to the length of the critical section. We can still use these protocols to implement $L(T^{[2]})$. Doing so allows us to merge phases $T^{[1]}$, $T^{[2]}$, and $T^{[3]}$ into a single phase with worst-case execution requirement $1.5 + 0.6 + 4.2 = 6.3$. The techniques that we present in Sec. 4 can then be used to implement $L(T^{[4]})$.

We now focus on deriving values for $B(T^{[i]})$ and $k(T^{[i]})$.

3.1 Skip Protocol (SP)

Under the SP, a lock request for ℓ that is found in $\mathsf{WAIT}(\ell)$ during the frozen interval is left in $\mathsf{WAIT}(\ell)$ and simply ignored until the requesting task is scheduled again. Doing this may reduce the task's blocking time by allowing it to retain its position in $\mathsf{WAIT}(\ell)$ at the cost of more complexity and overhead in managing $\mathsf{WAIT}(\ell)$. In particular, dequeuing from $\mathsf{WAIT}(\ell)$ will no longer be a constant-time operation.

Due to the FIFO ordering of $\mathsf{WAIT}(\ell)$, starvation of a request is not possible, which suggests that (R1) is easily satisfied as long as the execution cost of each critical section is less than a quantum. To satisfy (R2), the longest critical section guarded by ℓ, $\max\{T.C(\ell) \mid T \in \tau\}$, must be *strictly shorter* than the length of the frozen interval, $1 - t_F(\ell)$. (Recall that the critical section must complete *before* the quantum boundary.)

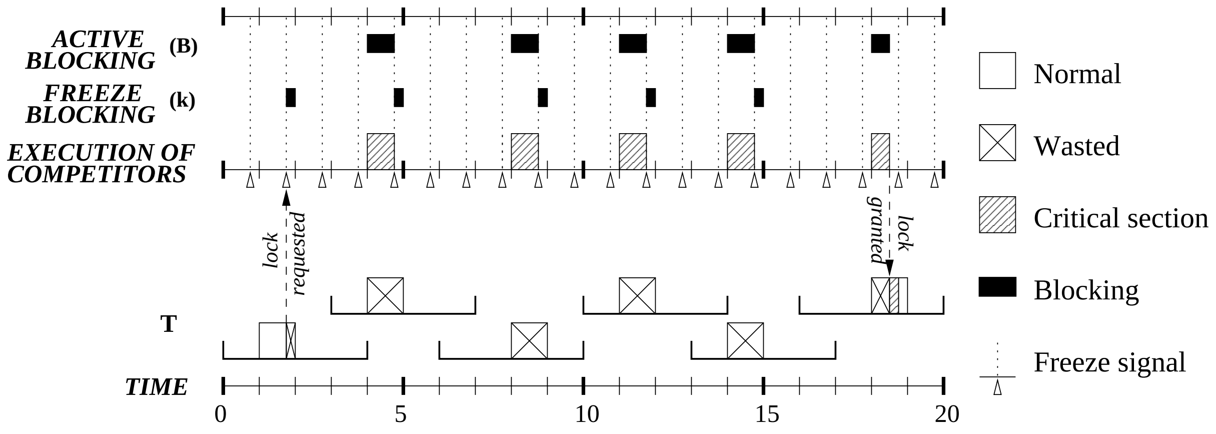


Figure 3: Worst-case blocking under the SP for a lock-requesting phase $T^{[i]}$ where $T.I(L(T^{[i]})) = 4.239$ and $t_F(L(T^{[i]})) = 0.9$. T's request is initiated at the start of a frozen interval and competing requests execute only when T is scheduled and never within a frozen interval. The blocking of each form experienced by T is shown with black bars at the top of the figure. The number of frozen intervals crossed (while scheduled) is $k(T^{[i]}) = 1 + \lfloor T.I(L(T^{[i]}))/t_F(L(T^{[i]}))\rfloor = 1 + \lfloor 4.71 \rfloor = 5$.

Hence, we have the following bounds on $t_F(\ell)$.

$$0 < t_F(\ell) < 1 - \mathsf{max}\{\ T.C(\ell) \ \mid\ T \in \tau\ \} \tag{3}$$

(The $0 < t_F(\ell)$ inequality is a trivial condition that is needed to ensure that the lock-granting mechanism of ℓ is not perpetually disabled.) To ensure the best possible performance in terms of both blocking overhead and responsiveness, $t_F(\ell)$ should always be set to the *largest* value that satisfies the above condition.

Theorem 1 *Under the SP, if $L(T^{[i]}) \neq \emptyset$, then the worst-case execution requirement of $T^{[i]}$, including blocking overhead, is no more than $T^{[i]}.e + B(T^{[i]}) + k(T^{[i]}) \cdot (1 - t_F(L(T^{[i]})))$ where $B(T^{[i]}) = T.I(L(T^{[i]}))$ and $k(T^{[i]}) = 1 + \lfloor T.I(L(T^{[i]}))/t_F(L(T^{[i]}))\rfloor$.*

Proof: Due to FIFO queueing, $T.I(L(T^{[i]}))$ is a trivial upper bound on the active blocking experienced by $T^{[i]}$. In the worst case, competitors of T execute for at least $t_F(L(T^{[i]}))$ time within each quantum in which T is scheduled, and hence do not release the lock before the start of the frozen interval. In this case, which is depicted in Fig. 3, T still crosses no more than $\lfloor T.I(L(T^{[i]}))/t_F(L(T^{[i]}))\rfloor$ quantum boundaries (while scheduled) before $L(T^{[i]})$ is granted. (Since the frozen interval in the time slot in which T is granted the lock is not counted, we use a floor expression rather than a ceiling.) In addition, if T's request is initiated at the start of a frozen interval, then T is blocked by an additional frozen interval. Hence, we must add one to the previous expression to account for this possibility. □

The previous analysis can be improved for systems that satisfy stronger restrictions, such as the following.

(R3) If a scheduled task T has a request pending for ℓ at the start of a quantum, then T's critical section is guaranteed to complete within that quantum.

Notice that (R3) is a stronger requirement than both (R1) and (R2). Unlike (R1) and (R2), this restriction bounds both the duration of individual critical sections and contention for the lock ℓ. The measurements of Ramamurthy, cited earlier, suggest that (R3) can be expected to hold in many cases.

For some lock ℓ to satisfy (R3), the total processing time required by any $M-1$ requests for ℓ, *i.e.*, $\mathsf{max}\{\ T.Q^1(\ell)\ \mid\ T \in \tau\ \}$, must be strictly less than the available per-slot processing time, $t_F(\ell)$. Hence, we have the following sufficient condition. ((3) provides the upper bound.)

$$\mathsf{max}\{\ T.Q^1(\ell)\ \mid\ T \in \tau\ \} < t_F(\ell) < 1 - \mathsf{max}\{\ T.C(\ell)\ \mid\ T \in \tau\ \} \qquad (4)$$

The following theorem generalizes Theorem 1 by assuming a bound on $t_F(\ell)$ that generalizes (4). Unfortunately, some systems may not satisfy the lower bound in (4). However, for such systems it might be possible to use a less-strict lower bound and still get smaller $B(T^{[i]})$ and $k(T^{[i]})$ values than are provided by Theorem 1. In Theorem 2, the strictness of the $t_F(\ell)$ lower bound is determined by the parameter m, where increasing m relaxes the bound. Condition (4) corresponds to the $m = 1$ case while Theorem 1 represents the limiting case in which $m \to \infty$.

Theorem 2 *Under the SP, if* $L(T^{[i]}) \neq \emptyset$ *and* $\frac{1}{m}\mathsf{max}\{\ T.Q^m(L(T^{[i]})\ \mid\ T \in \tau\ \} < t_F(L(T^{[i]}) < 1 - \mathsf{max}\{\ T.C(\ell)\ \mid\ T \in \tau\ \}$ *for some integer* $m \geq 1$, *then the worst-case execution requirement of* $T^{[i]}$, *including blocking overhead, is no more than* $T^{[i]}.e + B(T^{[i]}) + k(T^{[i]}) \cdot (1 - t_F(L(T^{[i]})))$ *where* $B(T^{[i]}) = Q^{m+1}(L(T^{[i]}))$ *and* $k(T^{[i]}) = m$.

Proof: Rewriting the lower bound on $t_F(L(T^{[i]})$ from the theorem statement produces $\mathsf{max}\{\ T.Q^m(L(T^{[i]})\ \mid\ T \in \tau\ \} < m \cdot t_F(L(T^{[i]})$. This condition, informally, states that the amount of processing time required by the competitors of $T^{[i]}$ across m quanta cannot consume all of the available execution time in those quanta. Hence, T is guaranteed to receive the lock within the mth full quantum that it receives after issuing the lock request, if not sooner. Including the quantum in which T issues the lock request, T can be actively blocked across at most $m+1$ quanta. Hence, $B(T^{[i]}) = Q^{m+1}(L(T^{[i]}))$ is a sufficient upper bound on active blocking. In addition, T may be blocked due to the frozen interval in each of these quanta except for the one in which the lock is granted to T. Therefore, at most m frozen intervals are crossed by T (while scheduled). $\square$

3.2 Rollback Protocol (RP)

Under the RP,[5] a request for lock ℓ that is found in $\mathsf{WAIT}(\ell)$ during the frozen interval is discarded and the requesting task must then re-issue it later. Although this protocol does not carry requests across quantum boundaries, it also sacrifices the guarantee that a lock-requesting task will be blocked by at most one request

[5]The term *rollback* refers to the fact that a requesting task's program counter may need to be set back so that it will re-issue a discarded request. This should not be confused with the "undo" processes commonly found in transaction-based systems.

of each competitor. As a result, preventing starvation (*i.e.*, guaranteeing (R1)) is more difficult under the RP. For this reason, we only consider the use of the RP when (R3) holds.

Theorem 3 *Under the RP, if $L(T^{[i]}) \neq \emptyset$ and $L(T^{[i]})$ satisfies (R3), then the worst-case execution requirement of $T^{[i]}$, including blocking overhead, is no more than $T^{[i]}.e + B(T^{[i]}) + k(T^{[i]}) \cdot (1 - t_F(L(T^{[i]})))$ where $B(T^{[i]}) = 2 \cdot T.Q^1(L(T^{[i]}))$ and $k(T^{[i]}) = 1$.*

Proof: This proof is almost identical to that of Theorem 2 when $m = 1$. However, a requesting task T may now be blocked twice by the same competitor (once per quantum). Hence, the worst-case execution requirement of competitors of T across two quanta, $T.Q^2(L(T^{[i]}))$, must be replaced with twice the worst-case execution requirement of competitors of T within a single quantum, $2 \cdot T.Q^1(L(T^{[i]}))$. □

3.3 Supertasking

We now explain how to adapt the previous theorems for systems using supertasks. Recall that supertasks can be used to prevent selected tasks from executing in parallel. Under the SP and the RP, tasks can only block each other by executing within the same time slot. Hence, supertasks can be used to reduce blocking overhead in systems using either the SP or the RP.

Additional notation. Let $\mathcal{S}.C(\ell) = \mathsf{max}\{\ T.C(\ell) \ \mid\ T \in \mathcal{S}\ \}$ denote the worst-case duration of any phase of a component task of supertask $\mathcal{S}$ that requires lock ℓ. Let $T.Q^m_S(\ell) = \mathsf{maxsum}_{m(M-1)}\{\ \mathcal{S}.C(\ell) \ \mid\ T \notin \mathcal{S}\ \}$ (respectively, $T.I_S(\ell) = \mathsf{sum}\{\ U.C(\ell) \ \mid\ U \notin \mathcal{S}(T)\ \}$) denote the worst-case execution cost of all competitors of T requesting lock ℓ across m quanta (respectively, across any number of quanta). (Note that two component tasks within the same supertask cannot be competitors since they cannot execute in parallel.)

Sufficient conditions with supertasking The theorems in the previous subsections can be easily adapted for supertasking by simply replacing all occurrences of $T.Q^m(\ell)$ and $T.I(\ell)$ with $T.Q^m_S(\ell)$ and $T.I_S(\ell)$), respectively. However, $\sum_{\mathcal{S} \in \sigma} \mathcal{S}.w \leq M$ must also be satisfied due to (1), where each supertask weight $\mathcal{S}.w$ ensures the safety of the component task deadlines.

A simple partitioning heuristic. In previous work [13], we presented a simple heuristic for partitioning τ into supertasks so that contention for lock-free objects is reduced. This same heuristic can be applied when our timer-based protocols are used, with only minor adjustments, which are summarized below.

The heuristic orders locks by contention. Contention for lock ℓ is approximated by $U(\ell) = \sum_{T \in \tau} T.U(\ell)$ where $T.U(\ell) = (\sum_{L(T^{[i]})=\ell} T^{[i]}.e)/T.p$. We refer to $U(\ell)$ as the *lock utilization* of ℓ. After selecting the lock ℓ with the largest $U(\ell)$, the heuristic packs tasks requiring ℓ into supertasks in non-increasing order by $T.C(\ell)$. Although we could use $T.U(\ell)$ instead to do this, $T.C(\ell)$ is the better choice because Theorems 1–3 only consider the worst *single* request made by each task (due to the use of FIFO queues).

4 Long Critical Section Protocols

In this section, we present our server-based protocol, which is intended for cases in which the critical-section length restrictions presented in Sec. 3 are not satisfied. We developed our server-based protocol after first considering several inheritance-based schemes, which we found to be problematic in fair-scheduled systems. To motivate the use of servers, we briefly consider these schemes below and point out some of their limitations.

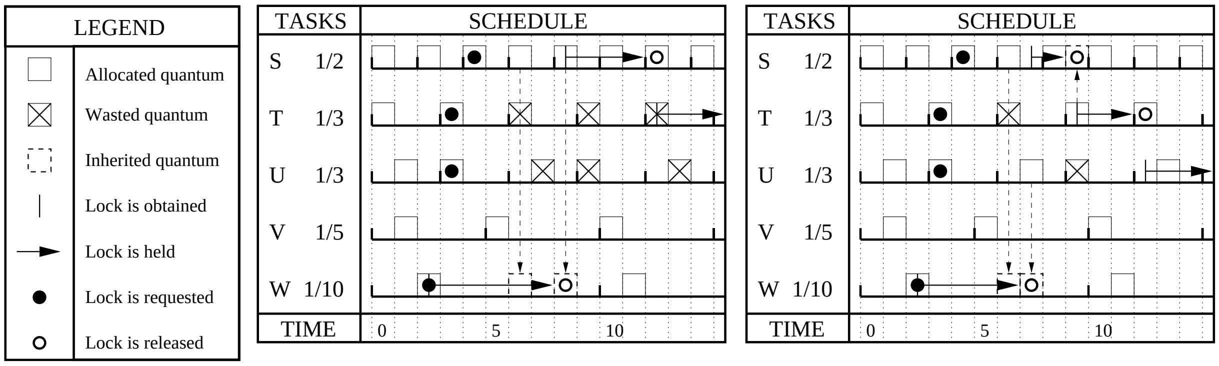
We consider only protocols that use statically-defined task weights here. Under a dynamic-weight protocol, a lock-holding task could speed its critical section by temporarily adding to its own weight the weights of tasks that it blocks. Unfortunately, Srinivasan and Anderson's work on dynamic task systems [24] implies that immediate weight changes may result in deadline misses in Pfair-scheduled systems. Although the durations of delays necessary to ensure the schedulability of the system are bounded, such delays in locking protocols would likely make worst-case analysis overly pessimistic, thereby negating the potential benefits. However, dynamic-weight protocols may perform better on average, making them an interesting alternative for systems that do not require hard real-time guarantees.

4.1 Inheritance Protocols

Under an inheritance-based protocol, a lock-holding task is allowed to *inherit* some of the scheduling parameters of a lock-requesting task that it blocks. For example, under the priority inheritance protocol [19], a lock-holding task T inherits the priority of a higher-priority task that it blocks. In the Pfair inheritance schemes considered here, a task T that inherits another task U's scheduling parameters is temporarily bound to U's state. Thus, an explicit distinction is made between scheduling states and the tasks that are bound to them. Usually a task T is bound to its own state, but when inheritance occurs, this binding may change. Specifically, T may become bound to another task U's state, or even to *both* its own state *and* U's state, leaving U temporarily bound to no state. For simplicity, we assume that each state has a pointer to the task that is currently bound to it and that the scheduler will ensure that each task may receive at most one quantum within each time slot, regardless of the number of state bindings.

Rate Inheritance Protocol (RIP). The RIP is based on the priority inheritance protocol [22]. Since allocation rates are proportional to weights, we can speed a lock-holding task's execution by letting it inherit the state of the heaviest-weight task that it blocks (if it indeed blocks a heavier task). Fig. 4(a) illustrates this technique. In time slot 2, task W requests and is granted the lock, which it does not release until it executes for two quanta. Tasks S, T, and U then request the lock in time slots 3 and 4, as shown. Each is suspended since the lock is being held by W. Under the RIP, W inherits T's scheduling state at time 4 and S's state at time 5. (Inheritance only occurs at quantum boundaries.) Due to this, W executes within the quanta allocated to S in time slots 6 and 8.[6]

[6]Observe that W is not allowed to execute in time slot 5 due to Property (Pf), even though the processor idles. It is actually trivial to modify the scheduler so that a processor never idles when a task is ready to execute. However, since this is not relevant



(a) Rate inheritance **(b) Allocation/deadline inheritance**

Figure 4: Two-processor Pfair schedules for a non-independent task set consisting of five tasks with weights 1/2, 1/3, 1/3, 1/5, and 1/10, respectively. **(a)** Rate inheritance. **(b)** Allocation (or deadline) inheritance.

Deadline Inheritance Protocol (DIP). We can improve upon the RIP by allowing a lock-holding task to use the highest-priority state available to it in each time slot. This guarantees that the task holding a lock ℓ executes whenever its own state or the state of any task that is blocked on ℓ is scheduled. However, this guarantee comes at the cost of more overhead due to frequent binding changes. Fig. 4(b) illustrates this protocol. In time slot 3, both T and U block on W's lock, making their states available to W. However, since all of these states, including W's state, are not eligible to execute due to Property (Pf), W is not allowed to execute in time slot 4. Similarly, S blocks on W's lock in time slot 4, making its state available to S. Again, W is not permitted to execute in time slot 5 since all available states are ineligible. At time 6, the states of S, T, and U all become eligible again and W binds to S's state since it has the highest priority (earliest pseudo-deadline). Since both S and T are scheduled in time slot 6, both states are again ineligible at time 7, prompting W to binds itself to U's state.

Allocation Inheritance Protocol (AIP). The AIP avoids frequent binding changes by allowing the lock-holding task to simply bind to *every* state that is available to it. This approach trades the overhead of frequent binding changes in the DIP for the overhead of managing a many-to-one binding. Since only the form of overhead changes between the DIP and the AIP, the two produce the same results.[7] For example, in Fig. 4(b), rather than checking the priority of each available state at the start of each time slot, W would simply have bound itself to T *and* U's states at time 4 and then to S's state also at time 5.

Under each of these inheritance protocols, the duration of blocking experienced by a lock-requesting task depends on the weights of its competitors. This is problematic for two reasons. First, identifying the worst-case blocking scenario is difficult, particularly for protocols like the DIP and the AIP. Although pessimistic upper bounds are easily derived, their use can result in considerable schedulability loss. Second, selecting

to our analysis, we ignore this issue.

[7]That is, the schedules produced are identical and each lock is granted and released at the same time in both schedules. There may be some variation with respect to which inherited quanta are utilized by lock-holding tasks.

task weights is complicated under these protocols because task weights determine the worst-case blocking, which in turn determines the minimum weights at which the task set is guaranteed to be schedulable. Hence, changing some task T's weight may require a compensatory weight change in all other tasks that share locks with T, which may then require another change in T's weight. In this way, the interdependence of weights makes the effect of a weight change difficult to predict, which is undesirable even if the weight selection algorithm's time overhead is reasonable.

4.2 Static Weight Server Protocol (SWSP)

We now turn our attention to the use of servers. In the server approach, each lock ℓ is implemented by a server task S that executes all critical sections guarded by ℓ in place of the requesting tasks. We assume that each server S has a statically-defined weight $S.w$. The use of static server weights implies that the duration of a requesting task's blocking depends *only* on $S.w$ and the length of competing critical sections. Hence, the SWSP does not suffer from the weight-interdependence problems noted above.

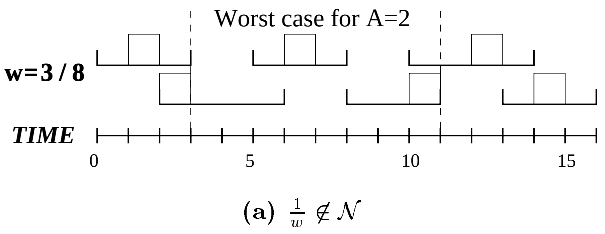
Detailed description. We let $S(\ell)$ denote the server task that implements lock ℓ. We assume that requests are still placed in a FIFO-ordered queue, $\mathsf{WAIT}(\ell)$. As long as $\mathsf{WAIT}(\ell)$ is non-empty, $S(\ell)$ releases its subtasks as early as is permitted under the IS task model. On the other hand, if $S(\ell)$ finds that $\mathsf{WAIT}(\ell)$ is empty, then it delays the release of its next subtask until $\mathsf{WAIT}(\ell)$ becomes non-empty again. We further assume that a requesting task T issues its request via an RPC, and hence is suspended until the response is received. We assume that RPC overheads are accounted for in T's execution cost.

Analysis. Before continuing, we define a few additional shorthand expressions. Let maxsum, $T.C(\ell)$, $T.Q^m(\ell)$, and $T.I(\ell)$ be defined as in Section 3. In addition, let $\delta(A, w)$ be as defined below, where $\mathcal{N}$ denotes the set of natural numbers. $\delta(A, w)$ is the shortest interval of time over which a task with weight w that does not experience IS delays is guaranteed to receive at least A quanta. Not surprisingly, $\delta(A, w)$ is one slot less than the worst-case span of any $A + 1$ consecutive windows, as shown in Fig. 5.

$$\delta(A, w) = \begin{cases} \left\lceil \frac{A+1}{w} \right\rceil & , A \geq 1 \wedge \frac{1}{w} \notin \mathcal{N} \\ \frac{A+1}{w} - 1 & , A \geq 1 \wedge \frac{1}{w} \in \mathcal{N} \\ 0 & , \text{otherwise} \end{cases} \tag{5}$$

Lemma 1 *A task T that initiates a locking phase $T^{[i]}$ within time slot t will receive a response from server $S(L(T^{[i]}))$ by time slot $t + 1 + \delta(\lceil T^{[i]}.e + T.I(L(T^{[i]}))\rceil, S(L(T^{[i]})).w)$.*

Proof: Since $T.I(L(T^{[i]}))$ is a trivial upper bound on the processing time required by competing requests, the server cannot execute for more than $\lceil T^{[i]}.e + T.I(L(T^{[i]}))\rceil$ quanta without completing T's critical section. Since the server runs continuously while T's request is pending, it follows that the server completes T's



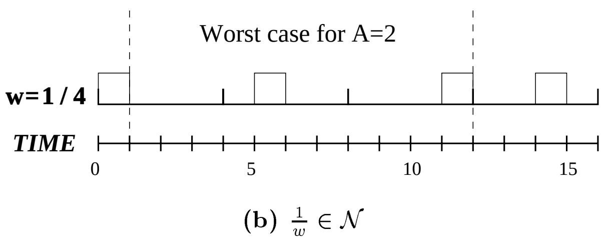


Figure 5: Marked intervals show the interval that defines $\delta(A,w)$ when **(a)** $w = 3/8$, $A = 2$, and $\delta(A,w) = \lceil (A+1)/w \rceil = 8$ and when **(b)** $w = 1/4$, $A = 2$, and $\delta(A,w) = (A+1)/w - 1 = 11$. As shown, $\delta(A,w)$ is one time slot shorter than the longest span of any $A+1$ consecutive windows.

critical section no later than $\delta(\lceil T^{[i]}.e + T.I(L(T^{[i]}))\rceil, S(L(T^{[i]})).w)$ time slots after receiving the request. One is then added to this previous value to account for the delay between the reception of the server's response and the start of the next time slot. □

Theorem 4 *Under the SWSP, a weight assignment of*

$$T.w = \frac{\sum\limits_{L(T^{[i]})=\emptyset} \lceil T^{[i]}.e \rceil}{T.p - \sum\limits_{L(T^{[i]})\neq\emptyset} (\delta(\lceil T^{[i]}.e + T.I(L(T^{[i]}))\rceil, S(L(T^{[i]})).w) + 1)}$$

that satisfies $0 < T.w \leq 1$ *is sufficient to ensure that all job deadlines of* T *are met in any Pfair schedule.*

Proof: We will assign a rational task weight to each task T of the form $T.a/T.b$ with the intention of associating the ith set of $T.a$ consecutive subtasks with the ith job of T. (Hence, the first subtask in each group is released when the associated job is released.) To ensure that T meets all job deadlines, we must show that no job of T can require more that $T.a$ quanta for local work and that all subtask deadlines occur before the job deadline. (It then follows by induction that all jobs will meet their deadlines.)

Fig. 6 illustrates the behavior of one job of a lock-requesting task T under the SWSP. As shown, server delays may cause remaining execution time within a quantum after a non-locking phase to be wasted. Hence, each non-locking phase $T^{[i]}$ may consume up to $\lceil T^{[i]}.e \rceil$ quanta. It follows that letting $T.a = \sum_{L(T^{[i]})=\emptyset} \lceil T^{[i]}.e \rceil$ ensures a sufficient allocation of processor time to each job of T.

When no IS delays are present, T's subtask windows will span exactly $T.b$ slots. If we let $\Delta(T_i)$ denote the release delay experienced by subtask T_i while waiting for a server response, then this span, with IS delays included, can be expressed as $T.b + \sum_{T_i \in J} \Delta(T_i)$, where J is the set of $T.a$ subtasks associated with the job in question. (Recall that subtasks and phases are different notions; a subtask is the unit of work associated with a window.) By Lemma 1, the request for phase $T^{[i]}$ receives a response after a delay of no more than $\delta(\lceil T^{[i]}.e + T.I(L(T^{[i]}))\rceil, S(L(T^{[i]})).w) + 1$. It follows that $\Delta(T_j) \leq \delta(\lceil T^{[i]}.e + T.I(L(T^{[i]}))\rceil, S(L(T^{[i]})).w) + 1$, where T_j is the first subtask released for the next non-locking phase. Hence, $\sum_{T_i \in J} \Delta(T_i) \leq \sum_{L(T^{[i]})\neq\emptyset} (\delta(\lceil T^{[i]}.e + T.I(L(T^{[i]}))\rceil, S(L(T^{[i]})).w) + 1)$ and letting $T.b = T.p - \sum_{L(T^{[i]})\neq\emptyset} (\delta(\lceil T^{[i]}.e + T.I(L(T^{[i]}))\rceil, S(L(T^{[i]})).w) + 1)$ ensures that all subtasks in J complete before the job deadline. □

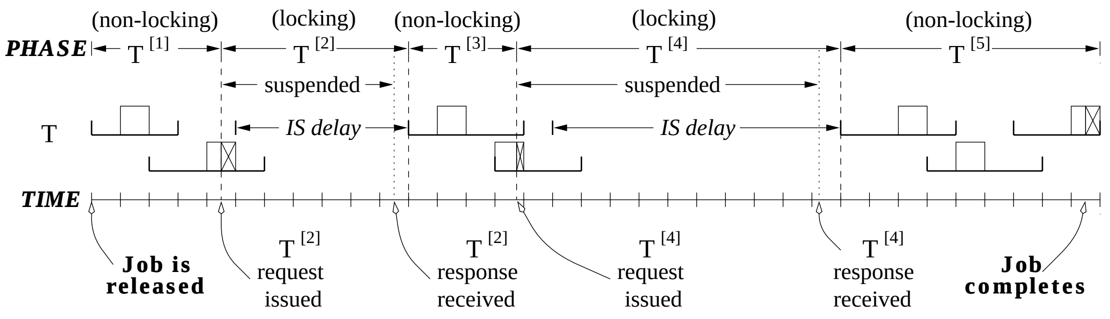


Figure 6: Behavior of a lock-requesting task T with weight $T.w = 7/19$ under the SWSP. The figure shows the release pattern for T's windows, where T has $T.n = 5$ phases per job. Phases $T^{[1]}$, $T^{[3]}$, and $T^{[5]}$ do not require locks, and hence are executed locally while phases $T^{[2]}$ and $T^{[4]}$ require locks and are executed remotely by the servers. Phase transitions are shown across the top of the figure and time is shown across the bottom. Arrows show when each RPC begins and ends. Unshaded boxes show where T executes locally while boxes containing X's denote unutilized processor time. Observe that remote execution results in an intra-sporadic delay between the non-locking phases, which causes T's windows to span an interval of 35 time slots instead of 19.

Assigning server weights. We now describe a simple algorithm for assigning weights to servers. First, suppose that we allot a fixed processor bandwidth, U_S, to all servers in the system. Since $U(\mathcal{L})/\sum_{\ell \in \rho} U(\ell)$ is a simple predictor of the fraction of processing time consumed by server $S(\mathcal{L})$ with respect to the processing time consumed by all servers, a simple and fair technique for assigning weights is to let $S(\mathcal{L}).w = \mathsf{min}(U_S \cdot (U(\mathcal{L})/\sum_{\ell \in \rho} U(\ell)), 1)$. In this approach, each server is assigned a weight that is proportional to the amount of processing time that it requires in the limit.

Supertasking. The SWSP also can be used in systems with supertasks. However, since preventing tasks from executing in parallel does not impact the worst-case server delay, the use of supertasks does not affect the worst-case blocking experienced under the SWSP.

5 Experimental Evaluation

In this section, we present an experimental evaluation of our protocols.

5.1 Experimental Setup

In each of our experiments, we determined the breakdown utilization of 20,000 random task sets on each of 2, 4, 8, 16, and 32 processors. We first randomly generated a lightweight, schedulable task set in which the total utilization was dominated by critical-section costs. We then systematically increased the utilization ($\sum_{T \in \tau} T.e/T.p$) by increasing the execution requirement of the non-locking phases of each task. (The costs of locking phases cannot be increased without also altering synchronization parameters such as $U(\ell)$.) We reported the highest schedulable utilization found before the first schedulability test failure. The average of these breakdown utilizations was then plotted against the cumulative lock utilization for the system

($\sum_{\ell \in \rho} U(\ell)$). Although these lock utilizations may appear small, recall that the length of a critical section tends to have far less impact on the total blocking time than delays due to preempted lock accesses.

We used the algorithm described at the end of Sec. 4 to assign weights to the SWSP servers. In the schedulability test, we checked schedulability with U_S set to each multiple of 0.05 in the range $(0, M/2]$. 0.05 was chosen to provide a fairly fine-grained coverage of the sample space. The $M/2$ upper bound reflects our belief that few, if any, randomly-generated task sets would require more than half of the system's capacity to be allocated to the servers.

The following ranges were chosen arbitrarily to generate random task sets: $|\rho| \in [5, 100]$, $|\tau| \in [25, 175 \cdot \log_2 M]$, $T.n \in [2, 12]$, and $T.p \in [200, 5000]$. In addition, critical-section lengths were chosen from the range $[0.001, UB]$, where UB is an experiment-dependent value (as explained later). Based on Ramamurthy's observations [21], we assumed that the simplest critical sections (*e.g.*, enqueuing to a shared queue) require at most a few microseconds on modern processors. Since a one millisecond quantum is typical, 0.001 quanta was chosen as a lower bound on critical-section lengths. To reflect the assumption that short critical sections are more common in practice, the distribution of critical-section lengths was skewed to favor smaller values.

Due to space constraints, we do not consider the use of the timer protocols with supertasks, Theorem 2 with $m > 1$, or the inheritance protocols in these experiments. Indeed, we are only able to present a few of the experiments that were conducted.

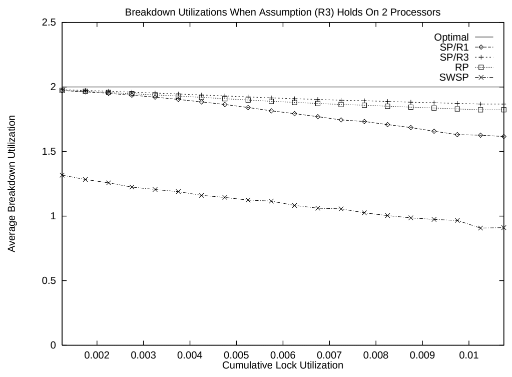
5.2 Experimental Results

We now present the results of three series of experiments. First, we evaluated the performance of all of our protocols with only short critical sections. Second, we evaluated the performance of the SP and SWSP when larger critical sections are present. Third, we evaluated the performance of hybrid systems in which each lock is implemented using the best available protocol (of those we have considered in this paper) as critical-section lengths increase.

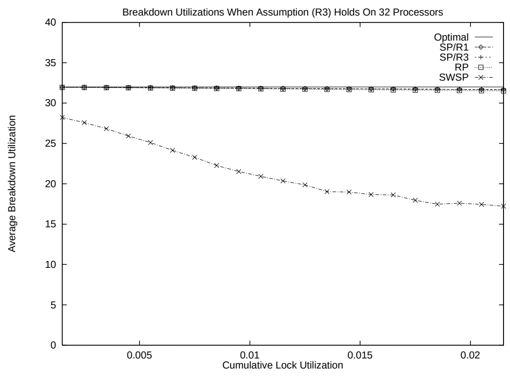
Short critical sections. In our first set of experiments, we considered only task sets containing short critical sections (generated using $UB = 0.25$) in which all locks satisfied (R3). We determined the breakdown utilization using each of the four schedulability conditions we have presented.

The results for the 2- and 32-processor cases are shown in Fig. 7(a)-(b), respectively. Due to space constraints, other cases are not shown, but resemble those presented here. In these graphs, SP/R1 and SP/R3 refer to the SP schedulability conditions developed assuming restrictions (R1)/(R2) and (R3), respectively.

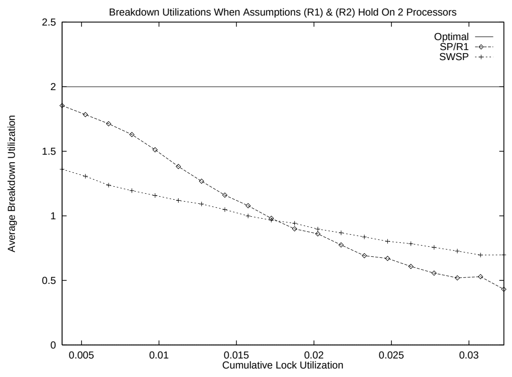
Not surprisingly, both the SP and RP perform very well, particularly in the $M = 32$ case in which the breakdown utilization is approximately 98.4% of the system utilization. (By *system utilization*, we mean the actual utilization divided by the number of processors.) The SWSP, on the other hand, performs significantly worse than the other protocols. The SWSP breakdown utilizations, expressed in terms of system utilization, range from 88.2% at best to around 45.5% at worst.



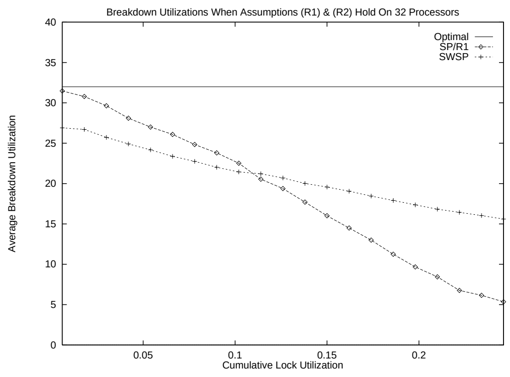
(a)



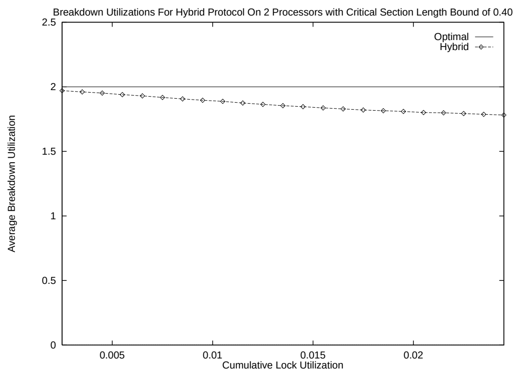
(b)



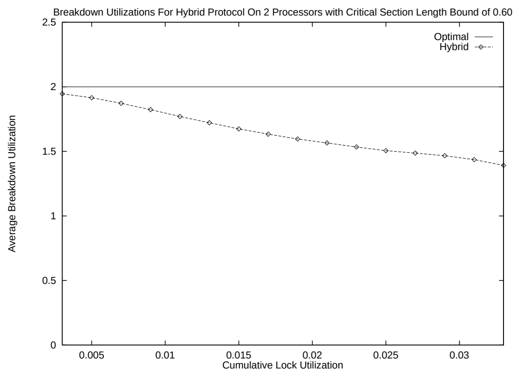
(c)



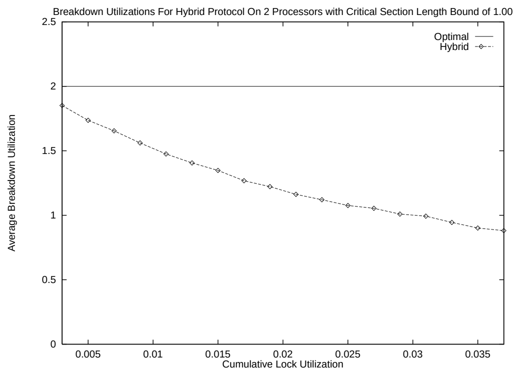
(d)



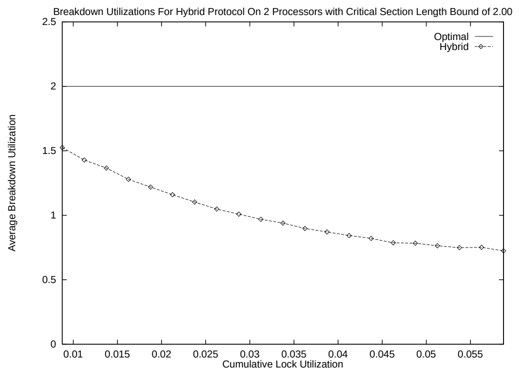
(e)



(f)



(g)



(h)

Figure 7: Breakdown utilization experimental results for **(a)** 2- and **(b)** 32-processor systems in which all locks satisfy (R3), **(c)** 2- and **(d)** 32-processor systems in which all locks satisfy (R1) and (R2), and 2-processor systems in which *UB* is set to **(e)** 0.4, **(f)** 0.6, **(g)** 1, and **(h)** 2.

Long critical sections. From the previous experiment, one might draw the mistaken conclusion that the SWSP serves no purpose. To dismiss this impression, we conducted a second experiment using only task sets generated using $UB = 1.00$ in which all locks satisfied (R1) and (R2). (Using larger UB values would accomplish nothing since the existence of a critical section of duration 1.0 or more guarantees that (3) cannot be satisfied, which precludes the use of the SP.) Whereas the previous experiment demonstrated that the SP and RP work well for short critical sections, this experiment demonstrates that the SWSP is more scalable than the SP when long critical sections are present.

The results of this experiment for 2- and 32-processor systems are given in Fig. 7(c)-(d), respectively. (Again, other cases are not included, but resemble those presented here.) As shown, the SP still outperforms the SWSP when the system-wide lock utilization is small. However, small system-wide lock utilization implies that lengthy critical sections are rare and that contention is low. As the system-wide lock utilization increases, the performance of the SP drops off quickly while the SWSP's performance drops off more gradually.

The timer-based protocols' lack of scalability with respect to critical-section lengths is not surprising. Recall that the length of the frozen intervals must be at least as long as the longest critical section. The SP should perform well, even under high contention, as long as the freeze interval requires only a small fraction of a quantum. As longer critical sections are introduced into the system, the freeze intervals expand and less processing time is available to critical sections within each quantum. In the extreme case, only one critical section will be allowed to execute within each quantum.

Hybrid systems. For the final experiment, we considered the effect of increasing UB over a series of runs. We considered all values of UB in the set $\{0.4, 0.6, 0.8, 1.0, 2.0\}$ in a 2-processor system. To determine the "best case" schedulability loss, we implemented a hybrid system that attempts to use the best of the protocols we have considered when implementing each lock. The results for $UB = 0.4$, 0.6, 1.0, and 2.0 are shown in Fig. 7(e)-(h), respectively. The performance improvement due to the timer-based protocols is evident when comparing these results to the performance of the SWSP alone in Fig. 7(a) and (c). (Note that the range of the x-axis differs in these graphs.)

Consider the breakdown utilizations that correspond to a system-wide lock utilization of approximately 0.025 in these graphs. As UB increases from 0.4 to 1.0, the breakdown utilization consistently decreases. For the $UB = 0.4$, 0.6, 0.8, and 1.0 runs, the breakdown utilization, expressed as a percentage of the system utilization, is 89.1%, 75.3%, 65.1%, and 53.8%, respectively. (The $UB = 0.8$ graph is not shown.) Interestingly, the performance does not degrade much between the $UB = 1.0$ and $UB = 2.0$ runs, which suggests that the SP and RP are being used infrequently in both runs.

6 Conclusion

In this paper, we have addressed the problem of providing support for lock-based synchronization in fair-scheduled multiprocessor systems. We have derived schedulability conditions for several protocols and experimentally evaluated them. Two of these protocols are optimized for systems in which critical sections are short relative to the length of a quantum, which is expected to often be the case in practice. For these protocols, we have explained how supertasks can be applied to reduce synchronization overhead further and have suggested a simple heuristic for assigning tasks to supertasks. These protocols differ from other locking protocols in that they focus on preventing the preemption of lock-holding tasks rather than bounding blocking due to such preemptions. As a result, blocking estimates depend on critical-section lengths alone and *not* on the scheduling of lock-holding tasks. We have also presented a more general-purpose server protocol that can be used when long critical sections prevent the efficient use of the optimized protocols. The protocols in this paper have the advantage that they can be used on a per-lock basis, and hence can be used together in the same system.

References

[1] J. Anderson and S. Ramamurthy. A framework for implementing objects and scheduling tasks in lock-free real-time systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 92–105. December 1996.

[2] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free objects. *ACM Transactions on Computer Systems*, 15(6):388–395, May 1997.

[3] J. Anderson and A. Srinivasan. Early-release fair scheduling. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, pages 35–43, June 2000.

[4] J. Anderson and A. Srinivasan. Pfair scheduling: Beyond periodic task systems. In *Proceedings of the Seventh International Conference on Real-Time Computing Systems and Applications*, pages 297–306, December 2000.

[5] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pages 76–85, June 2001.

[6] T. Baker. Stack-based scheduling of real-time processes. *Real-Time Systems*, 3(1):67–99, March 1991.

[7] S. Baruah, N. Cohen, C.G. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[8] S. Baruah, J. Gehrke, and C. G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 280–288, April 1995.

[9] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus fair scheduling: A proportional-share cpu scheduling algorithm for symmetric multiprocessors. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI)*, 2000.

[10] D. de Niz, L. Abeni, S. Saewong, and R. Rajkumar. Resource sharing in reservation-based systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 171–180. December 2001.

[11] P. Gai, G. Lipari, and M. di Natale. Minimizing memory utilization of real-time task sets in single and multi-process or systems-on-a-chip. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 73–83. December 2001.

[12] P. Holman and J. Anderson. Guaranteeing pfair supertasks by reweighting. In *Proceedings of the 22nd IEEE Real-time Systems Symposium*, pages 203–212. December 2001.

[13] P. Holman and J. Anderson. Object sharing in pfair-scheduled multiprocessor systems. To appear in *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, June 2002.

[14] G. Lamastra, G. Lipari, and Luca Abeni. A bandwidth inheritance algorithm for real-time task synchronization in open systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 151–160. December 2001.

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

[16] M. Caccamo and L. Sha. Aperiodic servers with resource constraints. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 161–170. December 2001.

[17] M. Moir and S. Ramamurthy. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In *Proceedings of the Twentieth IEEE Real-Time Systems Symposium*, pages 294–303, December 1999.

[18] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 116–123, 1990.

[19] R. Rajkumar. *Synchronization in real-time systems – A priority inheritance approach.* Kluwer Academic Publishers, Boston, 1991.

[20] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of the Ninth IEEE Real-Time Systems Symposium*, pages 259–269. 1988.

[21] S. Ramamurthy. A lock-free approach to object sharing in real-time systems. Dissertation, University of North Carolina at Chapel Hill. 1997.

[22] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time system synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[23] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. To appear in *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, May 2002.

[24] A. Srinivasan and J. Anderson. Fair on-line scheduling of dynamic task systems on multiprocessors. In submission, 2002.

A Summary of Notational Conventions

Notation	Definition
$\mathcal{N}$	Set of natural numbers
τ	Given set of synchronous periodic tasks
ρ	Given set of locks/guards used to synchronize τ
N	Size of τ
M	Number of processors
T	Synchronous periodic task (from τ)
$T.w$	Assigned Pfair weight of T
$T.p$	Integer period of T
$T.n$	Number of phases of T
$T^{[i]}$	ith phase of T
$T^{[i]}.e$	Worst-case execution cost of phase $T^{[i]}$
$L(T^{[i]})$	Lock/guard required by phase $T^{[i]}$
$T.e$	Worst-case execution cost of T $\left(T.e = \sum_{i=1}^{T.n} T^{[i]}.e\right)$
$T.C(\ell)$	Worst-case duration of any phase of T requiring ℓ
$T.Q^m(\ell)$	Worst-case execution cost of all competitors of T requesting lock ℓ across m quanta
$T.I(\ell)$	Worst-case execution cost of all competitors of T requesting lock ℓ
$B(T^{[i]})$	Worst-case active blocking experienced by $T^{[i]}$
$k(T^{[i]})$	Worst-case number of frozen intervals crossed during the lifetime of $T^{[i]}$
$T.U(\ell)$	T's utilization of lock ℓ
T_i	ith subtask of T
$\omega(T_i)$	window of T_i
$r(T_i)$	pseudo-release of T_i (first time slot in window)
$d(T_i)$	pseudo-deadline of T_i (last time slot in window)
σ	partitioning of τ into supertasks
$\mathcal{S}$	Supertask (one element of σ)
$\mathcal{S}.w$	Pfair weight assigned to $\mathcal{S}$
$\mathcal{S}(T)$	Supertask that has task T as a component task
$\mathcal{S}.C(\ell)$	Worst-case duration of any phase that requires lock ℓ of a component task of supertask $\mathcal{S}$
$T.Q^m_S(\ell)$	Same as $T.Q^m(\ell)$, but for systems using supertasks
$T.I_S(\ell)$	Same as $T.I(\ell)$, but for systems using supertasks
$\delta(A, w)$	Smallest interval length over which a task with weight w is guaranteed to receive A quanta

Notation	**Definition**
ℓ,$\mathcal{L}$	Lock/Guard (from ρ)
$U(\ell)$	Utilization of ℓ by task in τ
$\texttt{lock}(\ell)$	lock operation for ℓ
$\texttt{unlock}(\ell)$	unlock operation for ℓ
$\mathsf{WAIT}(\ell)$	FIFO-ordered wait queue of ℓ
FREEZE(ℓ)	Timer signal that disables ℓ's lock-granting mechanism
$t_F(\ell)$	Offset, relative to the start of a quantum, at which FREEZE(ℓ) occurs
$\mathsf{maxsum}_k\{A\}$	Maximum sum of any $\mathsf{min}(k,n)$-element subset of the multiset A
$S(\ell)$	Server task that executes critical sections guarded by lock ℓ
$S(\ell).w$	Static Pfair weight assigned to $S(\ell)$
U_S	Fixed bandwidth assigned to all servers