Locking under Pfair Scheduling

PHILIP HOLMAN
University of North Carolina at Chapel Hill
and
JAMES H. ANDERSON
University of North Carolina at Chapel Hill

We present several locking synchronization protocols 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. This is accomplished by exploiting the quantum-based nature of Pfair scheduling, which provides *a priori* knowledge of all possible preemption points. The second and more general protocol class is applicable to any system. For this class, we consider the use of a client-server model. We also discuss the viability of inheritance-based protocols in Pfair-scheduled systems.

Categories and Subject Descriptors: C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-Time and Embedded Systems*; D.4.1 [**Operating Systems**]: Process Management—*Multiprocessing*; *Mutual Exclusion*; *Scheduling*; *Synchronization*; J.7 [**Computer Applications**]: Computers in Other Systems—*Real time*

General Terms: Algorithms, Design

Additional Key Words and Phrases: Locking, Multiprocessor, Pfairness, Real-time, Scheduling, Semaphore, Supertask, Synchronization

1. INTRODUCTION

In recent years, there has been considerable interest in fair scheduling algorithms for multiprocessor systems [Anderson and Srinivasan 2000a; 2000b; 2001; Baruah et al. 1996; Baruah et al. 1995; Chandra et al. 2000; Moir and Ramamurthy 1999; Srinivasan and Anderson 2002]. 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 [Chandra et al. 2000].

One limitation of most prior work on multiprocessor fair scheduling algorithms is

Work supported by NSF grants CCR 9988327, ITR 0082866, CCR 0204312, and CCR 0309825.
Preliminary versions of some content appeared previously in [Holman and Anderson 2002b; 2002a].

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 [Anderson et al. 1997; Baker 1991; Rajkumar 1990; 1991; Rajkumar et al. 1988; Sha et al. 1990]. 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 [de Niz et al. 2001; Gai et al. 2001; Lamastra et al. 2001; Caccamo and Sha 2001].

In this paper, we consider the problem of incorporating lock-based synchronization into fair-scheduled multiprocessor systems. (In related work [Holman and Anderson 2002b; 2005], we showed how to incoporate *lock-free* synchronization within such systems.) The notion of fairness that we consider is the *Pfairness* constraint proposed by Baruah *et al.* [Baruah et al. 1996]. Under Pfair scheduling, each task is assigned a *weight*, which is the fraction of a single processor required by that task. Scheduling decisions are then made using a fixed-size scheduling quantum so that each task receives approximately the amount of processor time designated by its assigned weight. We also limit attention to the scheduling of *periodic* [Liu and Layland 1973] and *sporadic* [Mok 1983] tasks. A periodic (respectively, sporadic) task is invoked repeatedly to generate a sequence of identical *jobs*; consecutive invocations are separated in time by a given exact (respectively, minimum) delay. Despite this restricted focus, many of our results can be easily adapted to other fair scheduling algorithms and notions of recurrent execution as well.

Contributions of this paper. This paper consists of four contributions. First, we begin by deriving rules by which independent periodic and sporadic tasks can be supported in a *realistic* Pfair-scheduled system, *i.e.*, we assume that the quantum size is given rather than being arbitrarily selectable. This support provides for task suspensions and also forms a basis for the analysis presented later in the paper. Second, we consider the viability of using inheritance-based protocols in Pfair-scheduled systems. *Inheritance* occurs when a lock-holding task temporarily adopts characteristics of tasks that it blocks. Due to the effectiveness of inheritance on uniprocessors [Rajkumar 1990; 1991], it is an obvious alternative of interest. Third, we propose an optimized technique for supporting synchronization under quantum-based scheduling when critical sections are shorter than the quantum length. Finally, we propose a simple server-based protocol to support long critical sections. For each protocol, we present supporting analysis.

This remainder of the paper is organized as follows. After summarizing revelant background information and presenting basic results in Section 2, we present our rules for assigning weights to independent periodic and sporadic tasks in Section 3. These rules serve as a basis for the remainder of the results. We begin our discussion of synchronization by considering the viability of inheritance-based protocols in Section 4. In Section 5, we present two protocols designed for short critical sections. We then present our server-based protocol, which supports arbitrarily long critical sections, in Section 6. A simple experimental comparison of these protocol is then presented in Section 7. We conclude in Section 8.

2. BACKGROUND

In this section, we summarize background information that is related to the results presented herein. In addition, we present several basic results that drive the derivations presented later in the paper.

Periodic and sporadic tasks. Let τ denote a set of periodic [Liu and Layland 1973] and sporadic [Mok 1983] tasks to be scheduled on M processors. Each periodic and sporadic task T is characterized by four parameters: an *offset* $T.\phi$, a per-job *execution requirement* $T.e$, a *period* $T.p$, and a *relative deadline* $T.d$. Each time the task is invoked, a *job* is released that must complete within $T.d$ time units. The first invocation occurs at time $T.\phi$. Under the periodic (respectively, sporadic) task model, the next invocation occurs exactly (respectively, at least) $T.p$ time units after the previous invocation. Each job requires $T.e$ units of processor time to complete. We let $T = \mathbf{P}(\phi, e, p, d)$ (respectively, $T = \mathbf{S}(\phi, e, p, d)$) denote a periodic (respectively, sporadic) task T with $T.\phi = \phi$, $T.e = e$, $T.p = p$, and $T.d = d$.

Multi-phase representation. To represent jobs with critical sections and suspensions, we use a multi-phase representation. This representation decomposes each job J into a sequence of $J.k$ *phases*. (When the range of a phase index i is not explicitly given, the range $1 \leq i \leq J.k$ should be assumed.) Each phase $J^{[i]}$ is either an *execution* phase or a *suspension* phase. Since all jobs of a task are identical, $T^{[i]}$ will be used to denote the i^{th} phase of *any* job of task T.

An execution phase is described by two parameters. $T^{[i]}.e$ (respectively, $T^{[i]}.R$) denotes the *execution requirement* (respectively, *resource*) of phase $T^{[i]}$. $T^{[i]}.R$ indicates which (non-processor) shared resource[1] is required by $T^{[i]}$. If $T^{[i]}$ is *not* a critical section, then $T^{[i]}.R = \emptyset$. We let $T^{[i]} = \mathbf{C}(R, e)$ denote an execution phase $T^{[i]}$ with $T^{[i]}.R = R$ and $T^{[i]}.e = e$.

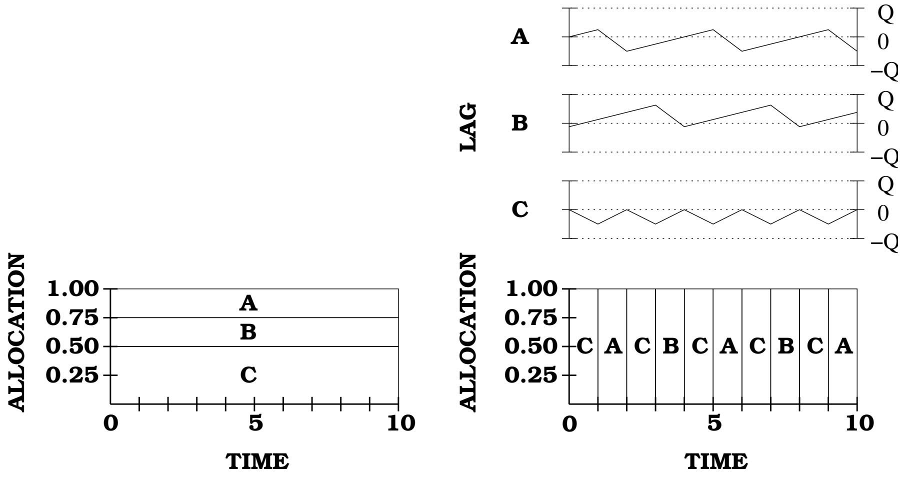
A suspension phase $T^{[i]}$ is characterized only by its maximum duration,[2] denoted $T^{[i]}.\theta$. Suspension phases are assumed to never occur consecutively. (Consecutive suspensions can be expressed as a single phase.) We let $T^{[i]} = \mathbf{I}(\theta)$ denote a suspension phase $T^{[i]}$ with $T^{[i]}.\theta = \theta$.

Pfair scheduling. Under Pfair scheduling, each task T is characterized by a *weight* $T.w$ in the range $(0, 1]$. Conceptually, $T.w$ is the fraction of a single processor to which T is entitled. We let $T = \mathbf{PF}(w)$ denote a Pfair task with $T.w = w$.

Time is subdivided into a sequence of fixed-length *slots*. To simplify the presentation, we use the slot length as the basic time unit, *i.e.*, slot i corresponds to the time interval $[i, i + 1)$. Within each slot, each processor may be allocated to at most one task. For instance, in Figure 1(b), task B is scheduled in slot 3, which

[1]This model does not support nested critical sections. Consequently, deadlock cannot occur. Adding support for nested critical sections and handling the complications introduced by them are topics for future work.

[2]Knowing the minimum duration is often useful also; however, for our purposes, the maximum duration is sufficient.



(a) Fluid scheduler

(b) Lag-based scheduler

Fig. 1. Sample schedules for $\tau = \{A, B, C\}$ where $A = \mathbf{PF}\left(\frac{1}{4}\right)$, $B = \mathbf{PF}\left(\frac{1}{4}\right)$, and $C = \mathbf{PF}\left(\frac{1}{2}\right)$. **(a)** Schedule produced by a fluid scheduler. **(b)** Schedule produced by a Pfair lag-based scheduler.

corresponds to the time interval $[3, 4)$. (The rest of this figure is considered in detail below.) Task migration is allowed. We let Q denote the *quantum* size, *i.e.*, the amount of processor time actually provided by each processor within each slot. In a real system, some processor time is unavoidably consumed in each slot by system activities, such as scheduling. We refer to such overhead as *per-slot* overhead. For example, if up to 10% of the processor time within each time slot is consumed by system activities, then $Q = 0.9$ (*i.e.*, 90% of the slot length). When practical overheads are ignored, as is commonly done in the literature, $Q = 1$ (*i.e.*, 100% of the slot length).

Pfair scheduling tracks the allocation of processor time in a fluid schedule; deviation is formally expressed as $lag(T, t)$, which is defined below.

$$lag(T, t) = fluid(T, 0, t) - received(T, 0, t) \tag{1}$$

In the above equation, $received(T, t_1, t_2)$ denotes the amount of processor time received by T over $[t_1, t_2)$, while $fluid(T, t_1, t_2)$ denotes the amount of processor time guaranteed by fluid scheduling over this interval. As explained in [Holman 2004], $fluid(T, t_1, t_2)$ is defined as shown below.[3]

$$fluid(T, t_1, t_2) = T.w \cdot (t_2 - t_1) \cdot Q \tag{2}$$

The above formula follows from the fact that each processor provides $(t_2 - t_1) \cdot Q$ units of processor time to tasks over $[t_1, t_2)$. Each task T is then entitled to a fraction $T.w$ of this quantity. (See [Holman 2004] for a more detailed explanation of fluid scheduling.) Using this notion of lag, the *Pfairness* timing constraint for a

[3]Because $Q = 1$ is commonly assumed, Q typically does not appear in similar formulas in the literature.

task T can be formally defined as shown below.

$$\text{for all } t,\ |\,lag(T,t)\,| < Q \tag{3}$$

Informally, T's allocation must always be within one quantum of its fluid allocation.

Figure 1(a) shows ideal (*i.e.*, $Q = 1$) fluid and Pfair uniprocessor schedules for a task set containing three Pfair tasks: $A = \mathbf{PF}\left(\frac{1}{4}\right)$, $B = \mathbf{PF}\left(\frac{1}{4}\right)$, and $C = \mathbf{PF}\left(\frac{1}{2}\right)$. In Figure 1(b), changes in each task's lag are shown across the top of the schedule.

Baruah *et al.* [Baruah et al. 1996] showed that a schedule satisfying (3) exists on M processors for a set τ of Pfair tasks if and only if the following condition holds.

$$\sum_{T \in \tau} T.w \leq M \tag{4}$$

Subtasks and windows. The use of quantum-based scheduling effectively subdivides each task into a sequence of quantum-length *subtasks*. Scheduling constraints, *e.g.*, (3), have the effect of specifying a *window* of slots in which each subtask must be scheduled. We let T_i denote the i^{th} subtask of task T, and let $\omega(T_i)$ denote the window of that subtask. Figure 2(a) shows the window within which each subtask of the task $\mathbf{PF}\left(\frac{3}{10}\right)$ must execute based on (3). For example, $\omega(T_2) = [3, 7)$. $\omega(T_i)$ extends from T_i's *pseudo-release*,[4] denoted $r(T_i)$, to its *pseudo-deadline*, denoted $d(T_i)$. In Figure 2(a), $r(T_2) = 3$ and $d(T_2) = 7$. A schedule satisfies Pfairness if and only if each subtask T_i executes in the interval $[r(T_i)\,, d(T_i))$. Finally, $b(T_i)$ is the number of slots by which the windows of T_i and T_{i+1} overlap, *i.e.*, $b(T_i) = d(T_i) - r(T_{i+1})$ [Anderson and Srinivasan 2000a].

Anderson and Srinivasan noted that the use of a rational weight results in a repeating, symmetric series of windows [Anderson and Srinivasan 2000a]. We refer to each occurrence of the series as a *cycle*. For instance, in Figure 2(a), each cycle consists of three windows that span ten slots. Two cycles are shown: the first spans $[0, 10)$ while the second spans $[10, 20)$. Cycles of a task T are defined by two parameters: the per-cycle execution requirement $\mathcal{E}(T)$ and the cycle period $\mathcal{P}(T)$. Given a weight $T.w = \frac{a}{b}$, where a and b are integers satisfying $b \geq a > 0$, $\mathcal{E}(T)$ and $\mathcal{P}(T)$ are defined by $\frac{a}{\mathsf{gcd}(a,b)}$ and $\frac{b}{\mathsf{gcd}(a,b)}$, respectively [Anderson and Srinivasan 2000a]. For instance, in Figure 2(a), $\mathcal{E}(T) = \frac{3}{\mathsf{gcd}(3,10)} = 3$ and $\mathcal{P}(T) = \frac{10}{\mathsf{gcd}(3,10)} = 10$.

Interest in $\mathcal{E}(T)$ and $\mathcal{P}(T)$ stems from the fact that these values define the placement of disjoint windows. More formally, these parameters satisfy Property WC, shown below.

Window Cycles (WC):

$$b(T_i) = \begin{cases} 0 \ , & \text{if } \mathcal{E}(T) \mid i \\ 1 \ , & \text{otherwise} \end{cases}$$

In the above property, $a \,|\, b$ holds when b is divisible by a. Informally, Property WC states that the last subtask in each cycle does not overlap with its successor, while all other subtasks within the cycle do. Hence, cycles do *not* overlap.

[4]The "pseudo" prefix avoids confusion with job releases and deadlines. This prefix will be omitted when the proper interpretation is clearly implied.

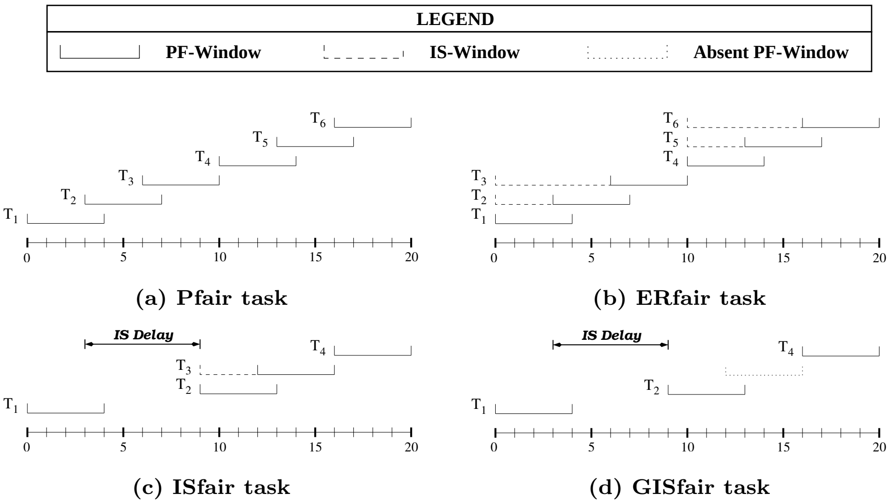


Fig. 2. The window layout for a task with weight $\frac{3}{10}$ is shown under the Pfair task model and its variants. **(a)** Normal window layout under the Pfair task model. **(b)** Early releasing has been used so that each grouping of three subtasks becomes eligible simultaneously. (In reality, each subtask will not be eligible until its predecessor is scheduled.) **(c)** Windows appear as in inset (b) except that T_2's release is now preceded by an intra-sporadic delay of six slots. (T_5 and T_6 are not shown.) **(d)** Windows appear as in inset (c) except that T_3 is now absent.

Pfair variants. Srinivasan and Anderson [Anderson and Srinivasan 2000a; Srinivasan and Anderson 2002] proposed three additional task models for use under Pfair scheduling. *Early-release* fairness (ERfairness) allows subtasks to execute before their pseudo-releases, provided that they are still prioritized by their pseudo-deadlines. *Intra-sporadic* fairness (ISfairness) extends ERfairness by allowing windows to be right-shifted (*i.e.*, delayed relative to their Pfair placement). However, the relative separation between each pair of windows must be at least that guaranteed under Pfairness. Under ISfairness, $b(T_i)$ is the *maximum* number of slots by which two windows can overlap, *i.e.*, the overlap when no delays occur. Finally, *generalized intra-sporadic* fairness (GISfairness) extends ISfairness by allowing subtasks to be omitted. Figure 2(b)–(d) illustrates these variants. Since the GIS model generalizes the ER and IS models, we restrict attention to the Pfair and GIS task models in the remainder of the paper. A GISfair task will be denoted by $\mathbf{GIS}(w)$.

Pfair schedulers. Several Pfair algorithms have been proposed, including PF [Baruah et al. 1996], PD [Baruah et al. 1995], PD^2 [Anderson and Srinivasan 2001], and EPDF [Anderson and Srinivasan 2000b; Srinivasan and Anderson 2003a]. Each of PF, PD, and PD^2 optimally schedules Pfair tasks, *i.e.*, its use will result in a Pfair schedule whenever (4) is satisfied. In addition, Anderson and Srinivasan proved that PD^2 correctly schedules ERfair, ISfair, and GISfair tasks whenever (4) holds. EPDF optimally schedules Pfair tasks only for systems of at most two processors [Anderson and Srinivasan 2000b]. Despite this, EPDF offers some practical advantages over the optimal algorithms, such as lower scheduling overhead.

In this paper, we consider only the *guarantees* provided by the scheduler and base our work on properties that follow from these guarantees. There are two primary benefits to abstracting the scheduler in this way. First, our results can be applied easily to both the optimal and sub-optimal Pfair schedulers. As demonstrated by Anderson and Srinivasan [Anderson and Srinivasan 2000b], sub-optimal policies, such as EPDF, are capable of providing fairness guarantees similar to, but weaker than, the Pfairness guarantee. Such relaxed fairness poses an interesting trade-off since weaker guarantees are often offset by practical gains, such as lower scheduling overhead. By enabling the use of our results under a variety of schedulers, we lay the foundation for a quantitative evaluation of this trade-off. Second, more scheduling policies will likely be proposed in the future. By developing a model for Pfair-like schedulers, we provide some forward compatibility with future work and try to avoid the need to revisit this issue each time a new scheduler is proposed.

To characterize scheduling guarantees, we use a four-parameter model, previously proposed by us in [Holman and Anderson 2003]. First, we let β_- (≥ 1) and β_+ (≥ 1) denote (real-valued) lower and upper *lag scalers*. These scalers are multiplied by $-Q$ and Q, respectively, to yield the actual lag bounds guaranteed by the scheduler, as shown below.

$$\text{for all } t,\ -Q \cdot \beta_- < lag(T,t) < Q \cdot \beta_+ \tag{5}$$

To simplify the presentation, we let

$$\beta = \beta_+ + \beta_-. \tag{6}$$

The constraint given by (5) generalizes (3), which corresponds to the $\beta_- = \beta_+ = 1$ case.

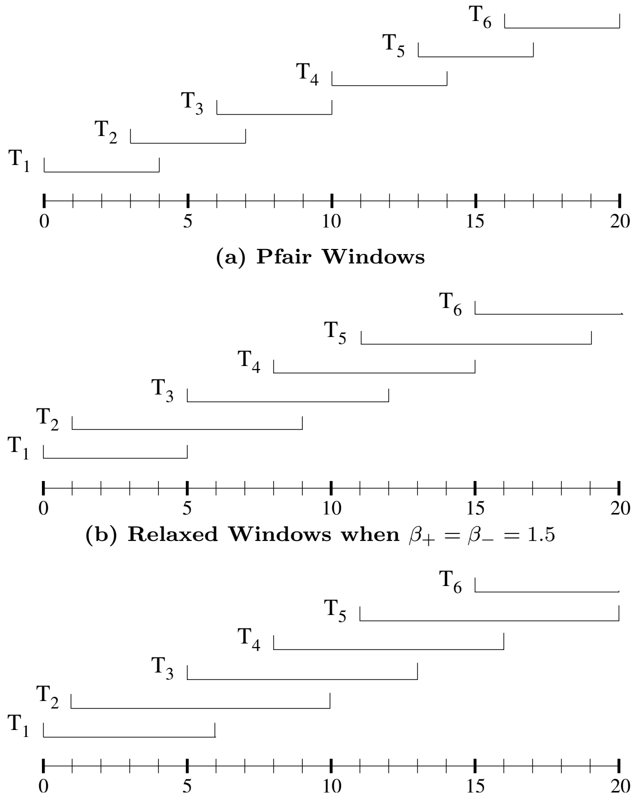
Relaxing lag bounds scales each subtask window. However, due to the use of quantum-based scheduling, windows are clipped to slot boundaries, resulting in non-uniform scaling. We refer to the windows defined by (5) as *relaxed* windows. Figure 3(b) shows the first six relaxed windows for a task with weight $\frac{3}{10}$ when $\beta_- = \beta_+ = 1.5$; Figure 3(a) shows the corresponding Pfair window layout. Notice that $\omega(T_2)$'s release occurs two slots earlier in Figure 3(b), while $\omega(T_3)$'s release occurs only one slot earlier.

The second parameter pair is ϵ_r and ϵ_d, which denote the number of slots by which each pseudo-release and pseudo-deadline, respectively, is extended (beyond its lag-based placement). More precisely, the scheduler treats a subtask with a relaxed window spanning $[t_r, t_d)$ as having the window $[t_r - \epsilon_r, t_d + \epsilon_d)$. Figure 3(c) shows the window layout obtained by $\beta_- = \beta_+ = 1.5$, $\epsilon_r = 0$, and $\epsilon_d = 1$. Notice that each deadline is extended by one slot, relative to Figure 3(b), due to ϵ_d. Such windows are called *extended* windows. For example, T_2 in Figure 3(c) has an extended deadline at time 10. We let

$$\epsilon = \epsilon_r + \epsilon_d. \tag{7}$$

Basic properties. We now state without proof basic properties of the scheduler model described above. (These properties are proved in an appendix.) All results apply to the case in which no IS delays occur.

The first theorem, shown below, provides formulas for determining the placement of extended windows. These formulas represent only the *guarantee* provided by the



(a) Pfair Windows

(b) Relaxed Windows when $\beta_+ = \beta_- = 1.5$

(c) Extended Windows when $\beta_+ = \beta_- = 1.5$, $\epsilon_r = 0$, and $\epsilon_d = 1$

Fig. 3. The first six windows of a task with weight $\frac{3}{10}$ are shown up to time 20. **(a)** Windows defined by Pfairness constraint. **(b)** Relaxed windows defined by $\beta_+ = \beta_- = 1.5$. **(c)** Extended windows defined by $\beta_+ = \beta_- = 1.5$, $\epsilon_r = 0$, and $\epsilon_d = 1$.

scheduler; we make no assumptions about how this guarantee is provided by the scheduler, beyond those already stated.

THEOREM 2.1. *When no IS delays occur, the following formulas define the placement of extended windows:*

$$r(T_i) = \left\lfloor \frac{i - \beta_+}{T.w} \right\rfloor - \epsilon_r \qquad d(T_i) = \left\lceil \frac{(i-1) + \beta_-}{T.w} \right\rceil + \epsilon_d.$$

The next lemma bounds the number of slots by which consecutive subtask windows can overlap.

LEMMA 2.2. *When no IS delays occur, each pair of consecutive subtasks T_i and T_{i+1} satisfy the inequality shown below.*

$$\left\lceil \frac{\beta - 2}{T.w} \right\rceil + \epsilon \le d(T_i) - r(T_{i+1}) \le \left\lceil \frac{\beta - 2}{T.w} \right\rceil + \epsilon + 1$$

If T_i is the last subtask in a cycle, then

$$d(T_i) - r(T_{i+1}) = \mathcal{B}(T) \stackrel{\text{def}}{=} \left\lceil \frac{\beta_- - 1}{T.w} \right\rceil + \left\lceil \frac{\beta_+ - 1}{T.w} \right\rceil + \epsilon.$$

The previous lemma introduces the new parameter $\mathcal{B}(T)$, which is the number of slots by which consecutive cycles of T overlap (when no IS delays occur). Recall that $\mathcal{B}(T) = 0$ under Pfairness.

The next lemma bounds the number of slots spanned by a sequence of n consecutive windows, which we refer to as an *n-span*. For instance, the interval $[3, 13)$ in Figure 3(a) is a 3-span since $r(T_2) = 3$ and $d(T_4) = 14$. In general, each n-span corresponds to an interval $[r(T_{i+1}), d(T_{i+n}))$ for some integer i.

LEMMA 2.3. *When no IS delays occur, every sequence of consecutive subtasks $T_{i+1}, \ldots, T_{i+n}$ satisfies the following:*

$$\left\lceil \frac{n + \beta - 2}{T.w} \right\rceil + \epsilon \le d(T_{i+n}) - r(T_{i+1}) \le \left\lceil \frac{n + \beta - 2}{T.w} \right\rceil + \epsilon + 1.$$

When T_{i+1} begins a cycle (i.e., $\mathcal{E}(T) \mid i$ holds) and subtask $i + n$ ends a cycle (i.e., $\mathcal{E}(T) \mid (i + n)$ holds),

$$d(T_{i+n}) - r(T_{i+1}) = \frac{n}{T.w} + \mathcal{B}(T).$$

Locking. Let Γ denote the set of all locks used by tasks in τ. A *lock-requesting* task T *issues* a request for a lock ℓ by invoking `RequestLock`. Until ℓ is granted, T's request is *pending*. Pending requests for each lock ℓ are assumed to be prioritized using a FIFO policy.[5] Once ℓ is granted to T, we refer to T as the *lock-holding* task until it completes the corresponding call to `ReleaseLock`. If a task U has a pending request for ℓ while T holds ℓ, then T is said to *block* U. Similarly, if tasks T and U both have pending requests for ℓ, then these requests are *competing* and T and U are *competitors*. Finally, we assume that the overhead of the `RequestLock` and `ReleaseLock` calls are factored into the execution requirements of the appropriate phases of each task.

To motivate the need for locking protocols under Pfair scheduling, consider a task T with a long critical section that always executes at its normal rate, *i.e.*, no attempt is made to speed its critical section. Let C denote the amount of processor time required by the critical section. Since T's average execution rate is given by $Q \cdot T.w$, the execution of its critical section will make the associated lock unavailable to other tasks for approximately $\frac{C}{Q \cdot T.w}$ time units for sufficiently long intervals. Hence, tasks with low weights can potentially make locks unavailable for *very* long durations, which suggests a need for techniques that speed the execution of critical sections.

[5]Locking protocols typically prioritize requests by the scheduling priorities of the requesting tasks. However, this is not practical under Pfair scheduling because priorities are not fixed during critical sections. We consider a FIFO prioritization because it facilitates analysis and avoids request starvation. Consideration of other request prioritizations is left as future work.

3. MAPPING INDEPENDENT TASKS

Before returning to the issue of locking, we first present rules for selecting a task weight based on the parameters of an independent periodic or sporadic task. More precisely, we explain how to map the requirements of an independent periodic or sporadic task T onto a GIS task U. These rules will also be used when accounting for synchronization in the sections that follow.

Motivation. Prior work (*e.g.*, [Anderson and Srinivasan 2000a; Baruah et al. 1996; Baruah et al. 1995]) considered simply letting $U.w = \frac{T.e}{T.p}$, under the assumption that $T.\phi = 0$ and $T.d = T.p$. This assignment is lacking for two reasons. First, such an assignment does not account for task suspensions. ISfairness provides some limited support in that suspensions that begin and end on slot boundaries can be modelled as IS delays. The rules that follow generalize support for suspensions. Second, the rule given above does not consider practical constraints, including the impact of Q. Indeed, this assignment assumes that the quantum size can be made arbitrarily small so that $T.e$ is a multiple of Q for all T. In reality, context-switching overhead imposes a practical lower limit on size of the scheduling quantum. The rules presented below include Q as an explicit parameter, and hence provide a more realistic assessment of the cost of supporting periodic and sporadic tasks under a Pfair scheduler.

Constraints. The rules presented here are based on the following constraints:[6]

(1) Each job of T is associated with a unique group of k consecutive cycles of U; each cycle is associated with only one job of T.
(2) The extended release of the first subtask associated with a job must occur on a slot boundary at or after the job's release.
(3) The extended deadline of the last subtask associated with a job must occur on a slot boundary at or before the job's deadline.
(4) All subtasks must satisfy the minimum separation imposed by the GIS task model.

If the above constraints are satisfied, then all jobs will meet their deadlines provided that a sufficient amount of processor time is allocated to each job while it is not suspended.

Without suspensions. First, consider the problem of scheduling a periodic task when all jobs are released on slot boundaries.

Lemma 3.1. *If T is an independent periodic task that never suspends and both $T.\phi$ and $T.p$ are integers, then each job of T will complete by its deadline when executed within the allocation of a GIS task U with weight*

$$U.w = \frac{\left\lceil \frac{T.e}{Q} \right\rceil}{\min\left(\lfloor T.d \rfloor - \mathcal{B}(U)\, , T.p\right)},$$

provided that $U.w \in (0, 1]$.

[6]These constraints are not necessary, *i.e.*, rules that do not satisfy these constraints may also work correctly.

PROOF. First, since no suspensions occur, associating $\left\lceil \frac{T.e}{Q} \right\rceil$ subtasks with each job is sufficient to ensure that the per-job execution requirement is satisfied. In the remainder of the proof, we construct a window layout that ensures that each job receives its associated time by its deadline, while respecting the four constraints given above. We then use the properties proved earlier to determine a weight capable of producing the desired layout. If a job completes while unused associated subtasks remain, then these subtasks can simply be marked absent.

Because $T.\phi$ and $T.p$ are integers,[7] it follows that $T.\phi + k \cdot T.p$ is an integer also for all integers $k \geq 0$. Hence, every job release coincides with a slot boundary. Consider a specific job J that arrives at time a. We let the extended release of the first subtask associated with the job occur at this time, which satisfies Constraint 2. Recall that the next job will be handled by a separate cycle. By Lemma 2.2, consecutive cycles of U can overlap by up to $\mathcal{B}(U)$. Hence, since the next job's first cycle begins at $a + T.p$, the extended deadline of the last subtask associated with J must occur at or before time $a + T.p + \mathcal{B}(U)$ to satisfy Constraint 4. In addition, J's deadline occurs at time $a + T.d$. It follows that this same extended deadline must occur at or before this time in order to satisfy Constraint 3. Therefore, the extended deadline must occur at or before $a + \min\left(\lfloor T.d \rfloor, T.p + \mathcal{B}(U)\right)$. Hence, the total window span of the cycle(s) associated with J must be at most $\min\left(\lfloor T.d \rfloor, T.p + \mathcal{B}(U)\right)$.

We can now use previous results to select a weight such that exactly the span given above is ensured. (A shorter span could be used instead, but would result in an unnecessarily high weight.) Recall that $\left\lceil \frac{T.e}{Q} \right\rceil$ subtasks must be associated with each job. By Lemma 2.3, a span of s can be ensured across one or more cycles consisting of n subtasks (combined) by letting $U.w = \frac{n}{s - \mathcal{B}(U)}$. Substituting $s = \min\left(\lfloor T.d \rfloor, T.p + \mathcal{B}(U)\right)$ and $n = \left\lceil \frac{T.e}{Q} \right\rceil$ into this formula establishes the lemma. □

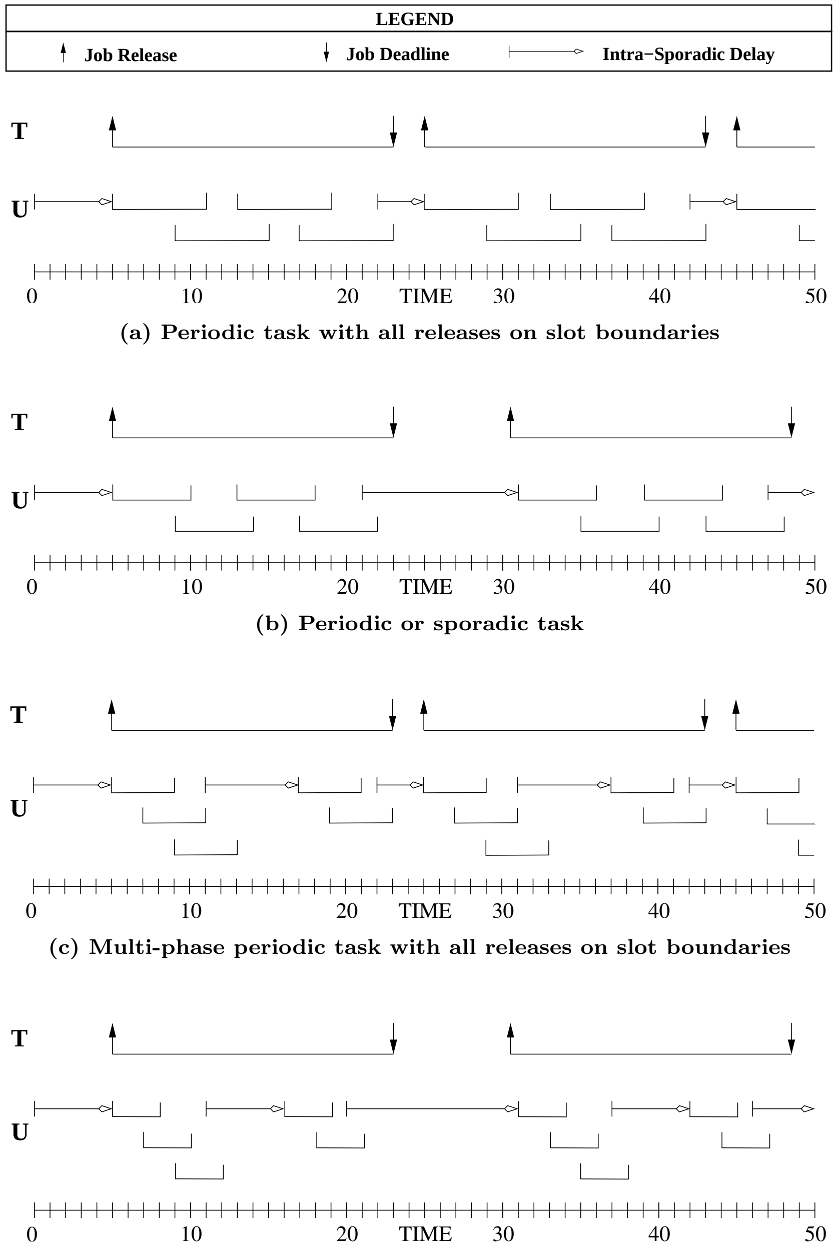
Figure 4(a) shows the application of Lemma 3.1 to the task $T = \mathbf{P}(5, 3.2Q, 20, 18)$ when the scheduler is described by $\beta_- = \beta_+ = 1$, $\epsilon_r = 0$, and $\epsilon_d = 1$. Hence, $\mathcal{B}(U) = 1$. The other insets, which consider this same scheduler with different tasks, are considered later. By Lemma 3.1, $U.w = \frac{\lceil 3.2 \rceil}{\min(\lfloor 18 \rfloor - 1,\ 20)} = \frac{4}{17}$. Notice how IS delays are used to ensure that subtask windows properly align properly with the associated job's interval. (Job releases and deadlines are depicted using up and down arrows, respectively.)

The above lemma highlights a problem that is common under rate-based scheduling: weight-assignment formulas are often functions of the task's weight. In this case, the $\mathcal{B}(U)$ term causes the problem. Applying the formula given in Lemma 2.2, the weight assignment given by Lemma 3.1 becomes

$$U.w = \frac{\left\lceil \frac{T.e}{Q} \right\rceil}{\min\left(\lfloor T.d \rfloor - \left(\left\lceil \frac{\beta_- - 1}{U.w} \right\rceil + \left\lceil \frac{\beta_+ - 1}{U.w} \right\rceil + \epsilon\right), T.p\right)}.$$

Notice that $U.w$ now occurs on both sides of the formula. Due to the ceiling operators in the right-hand side, these terms cannot be combined. In this case,

[7]Recall that the slot length is the basic time unit. Hence, in the context of this paper, a time value is a multiple of the slot length if and only if the value is an integer.



(d) Multi-phase periodic or sporadic task

Fig. 4. Examples of mapping a periodic or sporadic task T onto a GIS task U when the global scheduler is described by $\beta_- = \beta_+ = 1$, $\epsilon_r = 0$, and $\epsilon_d = 1$. Examples include **(a)** $T = \mathbf{P}(5, 3.2Q, 20, 18)$, **(b)** $T = \mathbf{S}(5, 3.2Q, 20, 18)$, **(c)** $T = \mathbf{P}(5, 3.2Q, 20, 18) : \mathbf{C}(\emptyset, 2.1Q)\,;\mathbf{I}(3.2)\,;\mathbf{C}(\emptyset, 1.1Q)$, and **(c)** $T = \mathbf{S}(5, 3.2Q, 20, 18) : \mathbf{C}(\emptyset, 2.1Q)\,;\mathbf{I}(3.2)\,;\mathbf{C}(\emptyset, 1.1Q)$.

this dependence can be avoided by using a scheduler for which $\beta_+ = \beta_- = 1$, as is done in our example above. However, in general, some parameters may require the use of iterative computation[8] or similar techniques. Since these techniques are a well-studied area of mathematics, we do not discuss them here.

The next lemma characterizes the cost of allowing jobs to be released off slot boundaries.

LEMMA 3.2. *If T is an independent periodic or sporadic task that never suspends, then each job of T will complete by its deadline when executed within the allocation of a GIS task U with weight*

$$U.w = \frac{\left\lceil \frac{T.e}{Q} \right\rceil}{\min\left(\lfloor T.d \rfloor - \mathcal{B}(U)\,, \lfloor T.p \rfloor\right) - 1},$$

provided that $U.w \in (0, 1]$.

PROOF. This proof closely resembles that of the previous lemma. Suppose a job release occurs t_Δ (< 1) time units before a slot boundary. In this case, the extended release of the first subtask associated with the job will not occur until the next slot boundary. Following identical reasoning to that given in the previous lemma, the span of the cycle(s) associated with each job can be at most $\min\left(\lfloor T.d - t_\Delta \rfloor\,, \lceil T.p - t_\Delta \rceil + \mathcal{B}(U)\right) \geq \min\left(\lfloor T.d \rfloor\,, \lfloor T.p \rfloor + \mathcal{B}(U)\right) - 1$. The remainder of the proof follows as in the previous lemma. $\square$

Figure 4(b) shows the application of Lemma 3.2 to $T = \mathbf{S}(5, 3.2Q, 20, 18)$. Applying the lemma yields $U.w = \frac{\lceil 3.2 \rceil}{\min(\lfloor 18 \rfloor - 1, 20) - 1} = \frac{4}{16} = \frac{1}{4}$, which implies that 4 cycles are associated with each job.

With suspensions. The next two lemmas generalize Lemmas 3.1 and 3.2 by adding support for suspension phases. The key idea is to insert a pessimistic IS delay that prevents the task from being granted processor time while it *may* be suspended. The proof of the following lemma explains this approach in more detail.

LEMMA 3.3. *If T is an independent multi-phase periodic task and both $T.\phi$ and $T.p$ are integers, then each job of T will complete by its deadline when executed within the allocation of a GIS task U with weight*

$$U.w = \frac{\sum\limits_{T^{[i]} \in \mathcal{C}(\emptyset)} \left\lceil \frac{T^{[i]}.e}{Q} \right\rceil}{\min\left(\lfloor T.d \rfloor - \mathcal{B}(U)\,, T.p\right) - \sum\limits_{T^{[i]} \in \mathcal{I}} \left(\left\lceil T^{[i]}.\theta \right\rceil + \left\lceil \frac{\beta - 2}{U.w} \right\rceil + \epsilon + 1 \right)},$$

provided that $U.w \in (0, 1]$.

PROOF. The lemma will be proved by showing that a per-job static window layout is capable of ensuring that each job completes on time. The idea is very similar to that used in the earlier proofs with one exception: in this case, we

[8]Iterative computations search for a non-trivial value x at which $x = f(x)$ for a given function $f(x)$. The process begins with a value x_0 and iterates using $x_{k+1} := f(x_k)$ until $x_{k+1} = x_k$ holds.

associate each subtask with only one *execution phase*. When a suspension follows an execution phase, we insert an IS delay after the last subtask window of that phase to ensure that the next phase does not begin until after the suspension has ended. We describe this approach in more detail below.

Consider the first phase $T^{[1]}$. For simplicity, assume $T^{[1]} \in \mathcal{C}(\emptyset)$. This phase must be associated with at least $\left\lceil \frac{T^{[1]}.e}{Q} \right\rceil$ subtasks. At some point at or before the extended deadline of the last subtask associated with this phase, denoted t_d, the suspension[9] that follows (if one indeed exists) is initiated. This suspension ends at or before time $t_d + T^{[2]}.\theta$. Hence, the first subtask associated with $J^{[3]}$ should not be released before time $t_d + \left\lceil T^{[2]}.\theta \right\rceil$ to ensure that allocated processor time is not wasted. Letting t_r denote the release of this successor subtask, this last restriction implies that $t_r = t_d + \left\lceil T^{[2]}.\theta \right\rceil$ is desired. By Lemma 2.2, $t_r \geq t_d - \left(\left\lceil \frac{\beta - 2}{T.w} \right\rceil + \epsilon + 1 \right)$ when the release is not delayed. It follows that a delay of $\left\lceil T^{[2]}.\theta \right\rceil + \left\lceil \frac{\beta-2}{T.w} \right\rceil + \epsilon + 1$ is sufficient to ensure the desired separation.

Using the above approach, we can insert IS delays to account for all suspensions. We now explain how a weight can be selected for U when this approach is used. As in Lemma 3.1, the windows associated with each job must span at most $\min\left(\lfloor T.d \rfloor, T.p + \mathcal{B}(U)\right)$ slots. However, this case differs from that in Lemma 3.1 in that this span reflects the span *after* inserting all IS delays. To select a task weight, we must determine the maximum span when no delays are present.

Inserting a delay has the effect of shifting all future releases and deadlines by the magnitude of the delay. Since the delay inserted for a specific suspension phase $T^{[i]}$ has a magnitude of $\left\lceil T^{[i]}.\theta \right\rceil + \left\lceil \frac{\beta-2}{U.w} \right\rceil + \epsilon + 1$ (as explained above), it follows that the total expansion of the span caused by all delays in a single job is given by $\sum_{T^{[i]} \in \mathcal{I}} \left(\left\lceil T^{[i]}.\theta \right\rceil + \left\lceil \frac{\beta-2}{U.w} \right\rceil + \epsilon + 1 \right)$. Hence, the span without these delays must be at most $\min\left(\lfloor T.d \rfloor, T.p + \mathcal{B}(U)\right) - \sum_{T^{[i]} \in \mathcal{I}} \left(\left\lceil T^{[i]}.\theta \right\rceil + \left\lceil \frac{\beta-2}{U.w} \right\rceil + \epsilon + 1 \right)$. The remainder of the theorem follows as in Lemma 3.1. □

Figure 4(c) shows the result of applying Lemma 3.3. In this example, $T = \mathbf{P}(5, 3.2Q, 20, 18)$ and each job consists of phases $\mathbf{C}(\emptyset, 2.1Q)$, $\mathbf{I}(3.2)$, and $\mathbf{C}(\emptyset, 1.1Q)$ (in the stated order). By Lemma 3.3, $U.w = \frac{\lceil 2.1 \rceil + \lceil 1.1 \rceil}{\min(\lfloor 18 \rfloor - 1, 20) - (\lceil 3.2 \rceil + 0 + 1 + 1)} = \frac{5}{11}$.

Lemma 3.4. *If T is an independent multi-phase periodic or sporadic task, then each job of T will complete by its deadline when executed within the allocation of a GIS task U with weight*

$$U.w = \frac{\sum_{T^{[i]} \in \mathcal{C}(\emptyset)} \left\lceil \frac{T^{[i]}.e}{Q} \right\rceil}{\min\left(\lfloor T.d \rfloor - \mathcal{B}(U), \lfloor T.p \rfloor\right) - 1 - \sum_{T^{[i]} \in \mathcal{I}} \left(\left\lceil T^{[i]}.\theta \right\rceil + \left\lceil \frac{\beta-2}{U.w} \right\rceil + \epsilon + 1 \right)},$$

provided that $U.w \in (0, 1]$.

[9]Since critical sections are not considered, each execution phase must be followed by a suspension phase. (Consecutive phases of the same type are assumed to be merged.)

Proof. This proof follows the same reasoning as the proofs of Lemmas 3.2 and 3.3. Specifically, the windows associated with each job must span no more than $\min(\lfloor T.d \rfloor, \lfloor T.p \rfloor + \mathcal{B}(T)) - 1$ slots by reasoning identical to that in Lemma 3.2. As in Lemma 3.3, this span must be shortened to account for inserted delays. The remainder of the proof follows as in Lemma 3.3. □

In Figure 4(d), the task T is described by $\mathbf{S}(5, 3.2Q, 20, 18)$ and each job consists of phases $\mathbf{C}(\emptyset, 2.1Q)$, $\mathbf{I}(3.2)$, and $\mathbf{C}(\emptyset, 1.1Q)$ (in the stated order). By Lemma 3.4, $U.w = \frac{\lceil 2.1 \rceil + \lceil 1.1 \rceil}{\min(\lfloor 18 \rfloor - 1, 20) - 1 - (\lceil 3.2 \rceil + 0 + 1 + 1)} = \frac{5}{10} = \frac{1}{2}$.

4. VIABILITY OF INHERITANCE-BASED PROTOCOLS

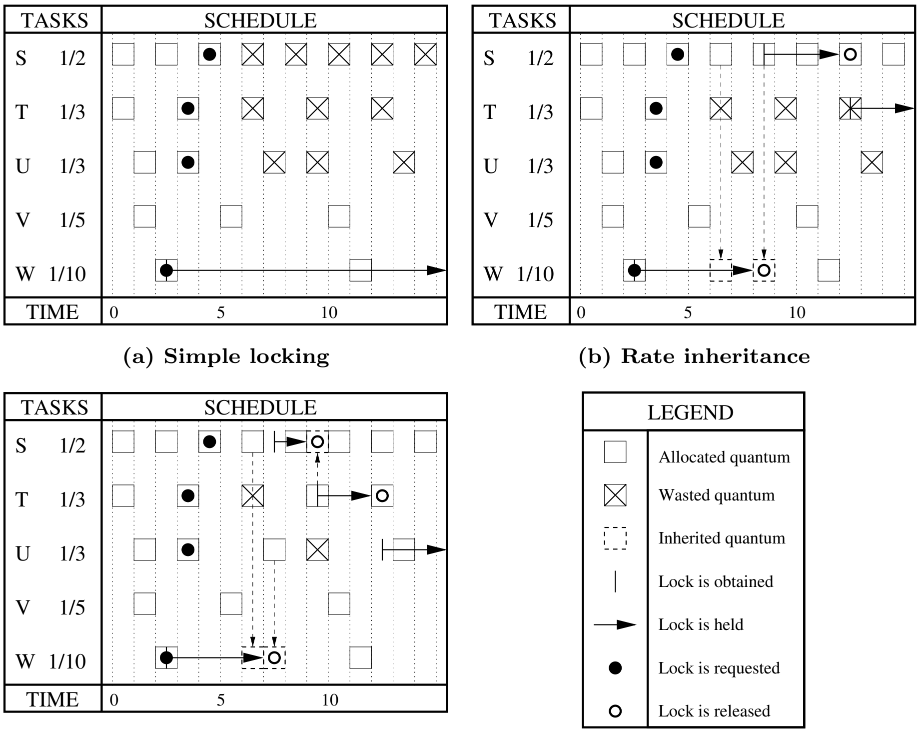
The most common approach to designing locking protocols is to rely on *inheritance* to speed the execution of lock-holding tasks [Baker 1991; Caccamo and Sha 2001; de Niz et al. 2001; Rajkumar 1990; 1991; Rajkumar et al. 1988; Sha et al. 1990]. Under such an approach, a lock-holding task takes on (*i.e.*, "inherits") the characteristics of tasks that it either blocks or has the potential to block. The lock-holding task then executes with these characteristics until releasing the associated lock. Unfortunately, the characteristics of Pfair scheduling complicate the use of such techniques (for reasons explained below). Consequently, the protocols presented in later sections are not based on inheritance. In this section, we discuss both the different forms of inheritance that are available under Pfair scheduling and the problems that can arise from their use.

4.1 Forms of Inheritance

Although conventional inheritance protocols are not directly applicable in Pfair-scheduled systems, many of the concepts underlying them are applicable. Since tasks are characterized only by weights, two obvious classes of inheritance-based protocols exist. The first class consists of *static-weight* inheritance protocols, under which weight changes are not permitted. Scheduling disruptions caused by blocking can be ameliorated under this restriction by decoupling the choice of which task *executes* in a slot, and which task is actually *charged* for the quantum. More specifically, critical sections can be executed more quickly by temporarily re-routing quanta from a blocked task to the task that is causing the blocking. The second class consists of *dynamic-weight* inheritance protocols, under which weight changes are permitted. Below, we describe some of the more obvious inheritance protocols and the characteristics of each.

Simple locking. Figure 5(a) provides a concrete example of how low-weight tasks can cause significant disruptions in Pfair systems. During slot 2, task W obtains a lock that is needed to execute a critical section requiring two quanta. When S, T, and U request this lock in slots 3–4, they are forced to wait until the lock is released by W. Due to W's low weight, this does not happen until sometime after slot 14. In the meantime, all quanta allocated to S, T, and U are wasted (because they are waiting for the lock).[10] We will use this same scenario to illustrate the protocols described below.

[10] In practice, IS delays would be used to prevent scheduling of the blocked tasks. This is not done in the examples given in this section so that the magnitude of disruptiveness will be apparent.



(a) Simple locking **(b) Rate inheritance**

(c) Deadline/allocation inheritance

Fig. 5. A series of two-processor Pfair schedules for a task set consisting of five tasks with weights $\frac{1}{2}$, $\frac{1}{3}$, $\frac{1}{3}$, $\frac{1}{5}$, and $\frac{1}{10}$, respectively, that may request locks. The insets illustrate the following static-weight locking scenarios: **(a)** simple locking, **(b)** rate inheritance, and **(c)** deadline/allocation inheritance.

Rate inheritance. Rate inheritance is based on the concept of priority inheritance [Sha et al. 1990]. Under this protocol, a lock-holding task T inherits the highest execution rate (*i.e.*, weight) of the tasks that it blocks (if that weight is higher than its normal weight). This protocol is easily implemented by swapping the association of weights to tasks, *i.e.*, the lock-holding task and the blocked task with the highest weight temporarily swap identities (scheduling parameters).

Figure 5(b) illustrates rate inheritance. In slot 3, W swaps with T and then with S in slot 4. As a result, W executes within S's quanta in slots 6 and 8.

One problem with rate inheritance is that the task with the highest weight may not have the highest priority at a given instant. For instance, in Figure 5(b), the priority of task U is higher than that of task S in slot 7. The next form of inheritance addresses this concern.

Deadline inheritance. Under deadline inheritance, a lock-holding task swaps identities with whichever blocked task has the highest priority at each time instant. This strategy improves upon rate inheritance, but at the expense of more swapping overhead. Specifically, both approaches may perform a swap when a task is initially blocked, but deadline inheritance may perform additional swaps each time a lock-holding task consumes a quantum.

Figure 5(c) illustrates deadline inheritance. In slot 3, W swaps with T. However, this swap is undone in slot 4 when W swaps with S. When W is executed in slot 6 (within S's quantum), U becomes the highest priority task, causing W to take on U's identity.

Allocation inheritance. An alternative form of inheritance that achieves the same result as deadline inheritance is allocation inheritance (which closely resembles bandwidth inheritance, as proposed for uniprocessor systems in [Lamastra et al. 2001]). To avoid the additional swapping overhead produced by deadline inheritance, the quanta allocated to *all* blocked tasks can be redirected to the lock-holding task for the duration of its critical section, *i.e.*, multiple sets of scheduling parameters may be mapped to the same task at a given instant. Although this approach reduces swapping overhead, it requires support for many-to-one mappings from parameter sets to tasks. In addition to the overhead required to maintain such a mapping, care must also be taken to ensure that each lock-holding task utilizes only one quantum in each slot. (Recall that parallel execution is not allowed.)

Figure 5(c) also illustrates the schedule as it would appear under allocation inheritance. In slot 3, the allocations of both U and V are redirected to W. S's allocation is then redirected to W in slot 4. Hence, at the start of slot 6, W is actually associated with four different sets of parameters (*i.e.*, those normally associated with S, U, V, and W). In slot 6, two of these sets are scheduled. However, W can only utilize one of the two quanta.

The potential for parallel allocation of quanta is a fundamental shortcoming of all forms of static-weight inheritance. To avoid this problem, subtask releases can be selectively postponed using IS delays. Unfortunately, such reactive behavior is difficult to characterize accurately when performing analysis, which necessitates the use of pessimistic assumptions. This added pessimism offsets, and may even outweigh, the provided benefits. Alternatively, a dynamic-weight scheme like that described next could be used.

Weight inheritance. Under weight inheritance, the lock-holding task adds to its weight the weight of each task that it blocks. (Weight inheritance also bears resemblance to bandwidth inheritance.) By changing its weight, the lock-holding task is able to effectively serialize its quanta in that it avoids the simultaneous allocation of multiple quanta. Weight inheritance is not illustrated in Figure 5 for reasons explained in the next section.

4.2 Problems

There are two primary problems that limit the effectiveness of inheritance-based schemes in Pfair systems. First, as explained in the previous section, dynamic-weight inheritance is the most effective form of inheritance in concept, due to the ability to serialize inherited processor time. Unfortunately, under Pfair scheduling (and likely under any other real-time rate-based approach), instantaneous weight changes must be *forbidden* to ensure schedulability [Srinivasan and Anderson 2003b]. Although improved reweighting techniques have been proposed [Anderson et al. 2003; Srinivasan and Anderson 2003a], these approaches either lack worst-case predictability [Anderson et al. 2003] or place additional restrictions on the system [Srinivasan and Anderson 2003a]. Accounting for delayed weight changes complicates worst-case analysis and necessitates the use of overly conservative weights, both of which offset the benefit of the protocol.

Unfortunately, even static-weight schemes are somewhat impractical due to the dependency between blocking overhead and task weights. Consider two tasks, T and U, that share a common lock. To account for the possibility that T blocks U, $U.w$ must be increased based on the worst-case blocking that can occur, which is determined by $T.w$. However, changing $U.w$ changes the worst-case duration for which U can block T, which necessitates a compensatory change in $T.w$. Hence, assigning weights under an inheritance protocol is an optimization problem. Finding an optimal solution is almost certainly an intractable problem.

Despite these issues, inheritance-based schemes remain an interesting avenue for future study, particularly for soft real-time systems. In addition, recent work [Block et al. 2005] on "fine-grained" reweighting techniques for Pfair scheduling shows considerable improvement over existing techniques. Unfortunately, due to significant differences between the system model considered in [Block et al. 2005] and that considered here, it is unclear whether fine-grained reweighting (or the concepts underlying it) can be used to produce an effective dynamic-weight locking protocol for hard real-time systems.

5. SUPPORTING SHORT CRITICAL SECTIONS

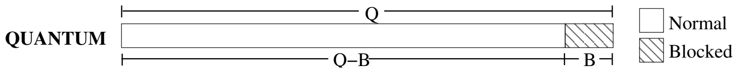
In this section, we present protocols for use with locks that guard short (relative to Q) critical sections. In experiments conducted by Ramamurthy [Ramamurthy 1997] on a 66 MHz processor, critical-section durations for a variety of common objects (*e.g.*, queues, linked lists, *etc.*) were found to 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, the protocols described in this section should be widely applicable. In the next section, we consider support for longer critical sections.

5.1 Concept

When some task T is granted a lock ℓ, each task U with a pending request for ℓ is necessarily delayed for the duration 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. As explained earlier, this is particularly problematic under Pfair scheduling because each task is executed at an approximately steady rate. Hence, if T has a low weight and its critical section is preempted before completing, then a lengthy delay is likely. Indeed, when the critical section is short, such a preemption effectively increases the critical-section duration by several orders of magnitude.

Fortunately, when using quantum-based scheduling with a sufficiently long quantum, the preemption of lock-holding tasks is avoidable. In particular, when T's critical-section duration is shorter than Q, 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 slot boundary. In practice, critical sections can be *forced* to start "sufficiently early" within a quantum by automatically blocking lock requests for critical sections that are at risk of being preempted. Whereas a task can only be blocked due to the unavailability of the requested lock in a traditional system, a task may be blocked due to either unavailability or the timing of the request under this approach.

To illustrate this approach, consider a lock request made by some task T, for which the associated critical section has worst-case duration e ($< Q$). The proposed approach introduces an interval of *automatic blocking* (of the lock request in question) at the end of each quantum, as illustrated below. We refer to this interval as a *blocking zone* and denote its length by B.



T's request is *safe* at any time instant not within that request's blocking zone, and *unsafe* otherwise. Additionally, T's request is said to be *active* if T is currently scheduled, and *inactive* otherwise. As long as $B \geq e$, T's critical section can be guaranteed to execute completely within a single quantum by simply granting the requested lock only if T's request is both safe and active. In this way, the nonpreemptivity inherent under quantum-based scheduling can be exploited to support short critical sections. Indeed, this approach is applicable under any scheduling approach that is based on the use of a quantum.

5.2 Zone Placement

There are several ways to apply the previous concept when designing a locking protocol. We refer to all protocols based on this concept as *zone-based* protocols. In this section, we briefly discuss some of the different rules for determining the placement of zones.

Request-based zones. The most aggressive approach is to provide supplementary information with each issued lock request (*i.e.*, details can be provided as input parameters to `RequestLock`). Whether the request is within its blocking zone at a given time can then be determined based upon this information and the state of the system. This approach introduces additional overhead into the lock-management algorithms, but also provides the most flexibility and can ensure that no critical section is delayed longer than necessary to ensure safety. For the latter reason, we focus on this approach when performing analysis later.

Group-based zones. Rather than associating a unique blocking zone with each request, each blocking zone can be associated with a group of critical sections. Under such an approach, B must be sufficiently long to prevent the preemption of each critical section in the group. The primary benefits of using such groupings are to simplify runtime accounting and to reduce the memory overhead of the protocol. To illustrate the former benefit, consider two of the most obvious groupings: group according to the requesting task or group according to the requested lock.

If a single blocking zone is used for all critical sections of a given task, then when a task is scheduled, an interrupt[11] that marks the start of its blocking zone can also be scheduled to occur on the assigned processor. Accounting for the overhead of the blocking zone is simplified in this case since each quantum requires only one such interrupt.

If, instead, a single blocking zone is used for all critical sections guarded by a given lock, then up to $|\Gamma|$ interrupts are needed in each quantum to signal the start of each lock's blocking zone. Although the use of more interrupts produces more overhead, the degree of unnecessary automatic blocking will likely be lower than when using task-based groups. This follows from the fact that locks are often used to synchronize operations of comparable complexity (and hence duration). On the other hand, the durations of critical sections of a given task may vary considerably.

Impact of timer precision. Real timers typically cannot generate interrupts at arbitrary times. Instead, expiration times (or delays) must be given as multiples of a timer-specific time unit. This unit defines the *precision*[12] of the timer.

Due to this practical restriction, it may be necessary to start blocking zones prematurely. Specifically, each zone that is initiated by an interrupt should be started at the latest possible timer expiration time that occurs before the ideal starting time of the zone. This conservative placement is always guaranteed to be within u of the ideal placement, where u denotes the timer's minimum time unit. Despite this, when u reflects a significant fraction of a quantum, timer error may be prohibitively high.

Indeed, there are several more issues relating to the use of realistic timers, including jitter and the handling of interrupts. Because we primarily focus on the use of request-based zones in the remainder of the paper, we forgo further discussion of these issues and leave them as topics for future work.

5.3 Requirements and Additional Notation

In the sections that follow, we present two protocols based on the notion of blocking zones. These protocols differ only in their handling of delayed requests found at slot boundaries. The first protocol maintains these requests, while the second protocol discards them, forcing the requesting tasks to re-issue them later. Hence, inactive requests may exist under the first protocol, but not under the second protocol. In this section, we begin by describing the base requirements of these protocols and by defining notational conventions used in the analysis presented later.

Requirements. The following requirements must be satisfied by a zone-based protocol.

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

(R2) Each critical section must execute completely within a single quantum.

(R2) requires that the zone associated with each request be at least the maximum duration of the request's critical section.

[11] We do not consider whether a hardware or software interrupt (or some other form of signaling mechanism) is most appropriate here, since this will likely depend on the target architecture.
[12] This is also called the *resolution* or *granularity* of the timer.

Notation. First, let $T^{[i]}.B$ denote the length of the blocking zone associated with the request of a locking phase $T^{[i]}$. This length is assumed to be sufficiently long to satisfy (R2). The following shorthand notations bound the worst-case blocking caused by each task within a single quantum.

—Let $T.\hat{e}(\ell) = \max\left\{ T^{[i]}.e \;\middle|\; T^{[i]}.R = \ell \right\}$. $T.\hat{e}(\ell)$ is the duration of the longest critical section of T requiring ℓ. This value is also an upper bound on the duration of blocking caused by T's requests for ℓ in a single quantum.

—Let $Q^m(T, \ell) = \mathsf{maxsum}_{m(M-1)} \{ U.\hat{e}(\ell) \mid U \neq T \}$. $Q^m(T, \ell)$ is an upper bound on the amount of time required to service competing requests of a single request of task T across m quanta while respecting a FIFO prioritization.

—Let $I(T, \ell) = \sum_{U \neq T} U.\hat{e}(\ell)$. $I(T, \ell)$ is an upper bound on the amount of time required to service all competing requests of a single request of task T while respecting a FIFO prioritization. Notice that $\lim_{m \to \infty} Q^m(T, \ell) = I(T, \ell)$.

In definition of $Q^m(T, \ell)$, $\mathsf{maxsum}_m \{a_1, \ldots, a_n\}$ denotes the maximum value produced by summing $\min(m, n)$ elements from the multiset $\{a_1, \ldots, a_n\}$. The following examples illustrate this notation:

—$\mathsf{maxsum}_2\{2, 2, 1\} = \max\{2 + 2, 2 + 1, 2 + 1\} = 4$;

—$\mathsf{maxsum}_4\{2, 2, 1\} = \max\{2 + 2 + 1\} = 5$;

—$\mathsf{maxsum}_0\{2, 2, 1\} = \max\{0\} = 0$.

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 while T's request is both safe and active. A lock cannot be granted during a blocking zone (*i.e.*, when T is unsafe), even if no competing requests are present. Blocking caused solely by blocking zones is called *automatic blocking*, as explained earlier.

5.4 Skip Protocol

Under the Skip Protocol (SP), delayed requests for locks are retained across slot boundaries, thereby ensuring that the FIFO prioritization is respected across slots. Due to this latter fact, the SP ensures that each competing task can block a request at most once. The primary cost of this approach is that the scheduler must either save and later restore the state of delayed requests at slot boundaries (*i.e.*, as the set of executing tasks changes), or simply ignore (*i.e.*, skip over) inactive requests when deciding which task will be granted a lock. In either case, this extra work translates into increased runtime overhead.

Due to FIFO prioritization and the fact that locks are not held across slot boundaries, request starvation is not possible. Hence, (R1) holds under the SP as long as $T^{[i]}.e < Q$ also holds for each locking phase $T^{[i]}$. To ensure that (R2) is satisfied, $T^{[i]}.B > T^{[i]}.e$ must also hold for each such locking phase to ensure that each critical section completes by the slot boundary that follows its start. Hence, we assume that the following relationship holds for each such locking phase $T^{[i]}$.

$$Q > T^{[i]}.B > T^{[i]}.e \tag{8}$$

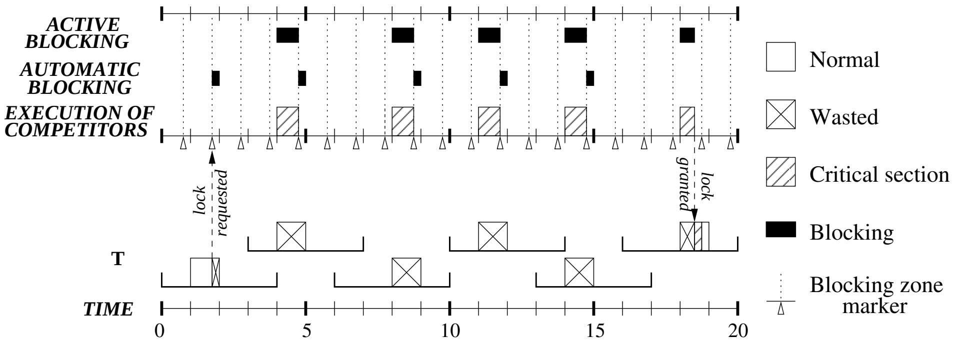


Fig. 6. Worst-case blocking under the SP for a lock-requesting phase $T^{[i]}$ when $I(T, \ell) = 4.239Q$ and $T^{[i]}.B = 0.1Q$. T's request is initiated at the start of a blocking zone and competing requests execute only when T is both scheduled and not within a blocking zone. The blocking of each form experienced by T is shown with black bars at the top of the figure. The number of blocking zones crossed (while scheduled) is $1 + \lfloor \frac{4.239Q}{Q-0.1Q} \rfloor = 1 + \lfloor 4.71 \rfloor = 5$.

(The $Q > T^{[i]}.B$ inequality is a trivial requirement that is needed to ensure that $T^{[i]}$ is considered safe at the start of each quantum.)

Basic analysis. To account for synchronization overhead, we present theorems and proofs that equate locking phases, assumed to execute under specified conditions, to non-locking phases. By substituting non-locking phases in this manner, we produce a description of a multi-phase *independent* task, which can then be used with Lemmas 3.1–3.4 to select an appropriate task weight. The next theorem handles the use of the SP.

THEOREM 5.1. *Given a phase $T^{[i]} \in \mathcal{C}(\ell)$, where ℓ is managed by the Skip Protocol and all blocking zones of ℓ satisfy* (8)*, phase $T^{[i]}$ is equivalent to*

$$\mathbf{C}\left(\emptyset, T^{[i]}.e + I(T,\ell) + \left(\left\lfloor \frac{I(T,\ell)}{Q - T^{[i]}.B} \right\rfloor + 1\right) \cdot T^{[i]}.B\right).$$

PROOF. Due to the FIFO prioritization being respected across slots, $I(T, \ell)$ is a trivial upper bound on the active blocking experienced by $T^{[i]}$. In the worst case, competitors of T execute for at least $Q - T^{[i]}.B$ time units within each quantum in which T is scheduled, and hence hold the lock until at least the start of T's blocking zone. In this case, which is depicted in Figure 6, T is preempted no more than $\left\lfloor \frac{I(T,\ell)}{Q-T^{[i]}.B} \right\rfloor$ times after issuing the request before ℓ is granted. (Since the blocking zone in the 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 blocking zone, then T is blocked by an additional blocking zone. Hence, the duration of phase $T^{[i]}$ is at most the sum of the durations of **(i)** $T^{[i]}$'s critical section, **(ii)** $I(T, \ell)$ units of active blocking, and **(iii)** $\left(\left\lfloor \frac{I(T,\ell)}{Q-T^{[i]}.B} \right\rfloor + 1\right) \cdot T^{[i]}.B$ units of automatic blocking. □

Improved analysis. Stronger restrictions, like (R3) given below, produce less overhead.

(R3) All active requests present at the start of a slot are serviced within that slot.

Notice that (R3) is a stronger requirement than both (R1) and (R2). Unlike (R1) and (R2), this restriction bounds the total time required by any combination of competing requests (up to M) that are serviced within the same slot. The measurements of Ramamurthy, cited earlier, suggest that (R3) can be expected to hold in many cases.

For (R3) to be satisfied with respect to a request for ℓ made by a locking phase $T^{[i]}$, the total processing time required by T's worst-case mix of $M-1$ competing requests for ℓ (*i.e.*, $Q^1(T,\ell)$), must be strictly less than the length of time over which T's request is safe (*i.e.*, $Q - T^{[i]}.B$). The condition given below is sufficient to ensure that (R3) holds for a lock ℓ.

$$\Big(\forall T : T.\hat{e}(\ell) > 0 : Q^1(T,\ell) \le Q - \max\Big\{\ T^{[i]}.B\ \Big|\ T^{[i]}.R = \ell\ \Big\}\Big) \tag{9}$$

The $T.\hat{e}(\ell) > 0$ constraint in the above condition simply restricts attention to tasks with critical sections that require ℓ.

Unfortunately, some systems may not satisfy the condition given in (9). However, for such systems, it is still possible to apply a less-strict condition to *individual* critical sections in order to produce blocking estimates tighter than those provided by Theorem 5.1. Theorem 5.2, given next, considers the condition

$$\frac{1}{m}Q^m(T,\ell) \le Q - T^{[i]}.B, \tag{10}$$

which focuses on a specific phase $T^{[i]}$ of a task T that requires ℓ. This condition generalizes (9) by providing an integer parameter m (≥ 1) that determines the strictness of the condition. Informally, m denotes the maximum number of blocking zones that can be crossed (while scheduled) before a task is granted the requested lock. Increasing m has the effect of weakening the condition. Condition (9) corresponds to the strictest case in which all critical sections satisfy (10) for $m = 1$. Theorem 5.1 then reflects the limiting behavior as $m \to \infty$.

THEOREM 5.2. *Given a phase $T^{[i]} \in \mathcal{C}(\ell)$ that satisfies* (10) *for some $m \ge 1$, where ℓ is managed by the Skip Protocol and all blocking zones of ℓ satisfy* (8)*, phase $T^{[i]}$ is equivalent to*

$$\mathbf{C}\Big(\emptyset, T^{[i]}.e + Q^{m+1}(T,\ell) + m \cdot T^{[i]}.B\Big).$$

PROOF. Condition (10) implies that $Q^m(T,\ell) \le m\left(Q - T^{[i]}.B\right)$ holds. Informally, this condition states that the amount of processing time required by the competitors of $T^{[i]}$ across m slots cannot consume all of the processor time available outside of T's blocking zones in those slots. Hence, T is guaranteed to receive the lock within the m^{th} slot in which it is scheduled after issuing the lock request, if not sooner. This scenario is depicted in Figure 7. Including the quantum in which T issues the lock request, T is actively blocked across at most $m+1$ quanta. Hence, the total duration of active blocking cannot exceed $Q^{m+1}(T,\ell)$ time units. In addition, T may be automatically blocked in each of these quanta except for

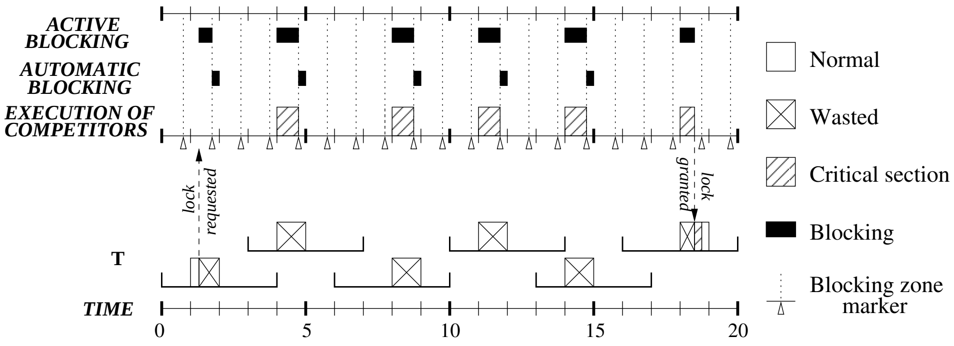


Fig. 7. Worst-case blocking under the SP for a lock-requesting phase $T^{[i]}$ when $Q^5(T, \ell) = 4.25Q$ and $T^{[i]}.B = 0.1Q$. Since $\frac{1}{5}Q^5(T, \ell) = 0.85Q$, (10) holds for $m = 5$. The resulting worst-case scenario is depicted. The number of blocking zones crossed (while scheduled) is 5 (*i.e.*, m), while active blocking occurs within 6 (*i.e.*, $m + 1$) quanta.

the last one, in which the lock is finally granted. Therefore, the request spans at most m blocking zones. Hence, the duration of phase $T^{[i]}$ is at most the sum of the durations of **(i)** $T^{[i]}$'s critical section, **(ii)** $Q^{m+1}(T, \ell)$ units of active blocking, and **(iii)** $m \cdot T^{[i]}.B$ units of automatic blocking. □

5.5 Rollback Protocol

Under the Rollback Protocol (RP),[13] delayed requests are discarded (failed) at slot boundaries and must be re-issued by the requesting tasks when they resume execution. Hence, FIFO prioritization only applies to requests issued within the same slot. The RP sacrifices the guarantee that a request will be blocked by at most one request of each competing task in order to avoid the additional overhead of maintaining requests across multiple slots. A side effect is that preventing starvation (*i.e.*, guaranteeing (R1)) is more difficult. To guarantee starvation avoidance, we restrict consideration of the RP to cases in which (R3) holds.

The next theorem characterizes the worst-case duration of a locking phase under the RP.

THEOREM 5.3. *Given a phase $T^{[i]} \in \mathcal{C}(\ell)$, where ℓ satisfies* (9)*, ℓ is managed by the Rollback Protocol, and all blocking zones of ℓ satisfy* (8)*, phase $T^{[i]}$ is equivalent to*

$$\mathbf{C}\Big(\emptyset, T^{[i]}.e + 2 \cdot Q^1(T, \ell) + T^{[i]}.B\Big).$$

PROOF. The case considered here is almost identical to the $m = 1$ case of Theorem 5.2. However, since a requesting task T can have its request rejected due to crossing a slot boundary, its request can potentially be blocked by the same group of competitors in each slot. From (R3) and this observation, it follows that T's

[13]The term *rollback* refers to the fact that a requesting task may need to re-issue a discarded request. This should not be confused with the "undo" processes commonly found in transaction-based systems.

request can be delayed due to active (respectively, automatic) blocking for no more than $2 \cdot Q^1(T, \ell)$ (respectively, $T^{[i]}.B$) time units. □

6. SUPPORTING LONG CRITICAL SECTIONS

In this section, we present a simple server-based protocol to support critical sections of arbitrary length. This protocol focuses on avoiding the problems associated with inheritance-based approaches. The resulting protocol is simple to use and analyze, but is expected to yield only mediocre performance. The primary purpose of this protocol is to provide a baseline performance measurement that can be used to evaluate the performance of the SP, RP, and other protocols. The specific goals of the design are listed below.

(1) Blocking durations should be independent of the weights of requesting tasks.
(2) Requesting tasks should not be required to change weights.
(3) Modifying task parameters should impact performance in a predictable way.
(4) The protocol should not introduce substantial per-slot overhead.

6.1 Static-weight Server Protocol

When using lock servers, a server task V executes all critical sections guarded by ℓ in place of the requesting tasks. Such servers are actually quite common in practice. In addition to implementing critical section and kernel calls, such special processes are often used to implement basic communication services, such as remote procedure calls (RPCs).

Issues. The primary issue when using server tasks is the scheduling of the server. Many approaches have been proposed to exploit the structure of the system in which the servers will execute. For instance, in fixed-priority systems, servers can be scheduled as normal tasks and inherit the priority of the requesting tasks while servicing requests. This policy, called *priority tracking*, addresses the potential for priority inversion[14] caused by the server's processing of requests on behalf of low-priority tasks. For example, kernel threads in the LynxOS use this approach [Lynx Real-time Systems 1993]. Similarly, in reservation-based systems (*e.g.*, see [Caccamo and Sha 2001; de Niz et al. 2001]), the reservation[15] of the requesting task can be passed to the server. This ensures that the processor time consumed by the server is properly charged against the requesting tasks.

To satisfy Goal 1, we consider only the use of server tasks with statically defined weights. When using servers, the duration of a locking phase is determined by the worst-case responsiveness of the associated server task. This responsiveness, in turn, depends on the scheduling of the server and the amount of processor time required to complete the operations. We refer to the server protocol considered here as the Static-weight Server Protocol (SWSP).

[14]A *priority inversion* occurs when a task is prevented from executing by a lower priority task.
[15]A *reservation* is a form of rate-enforcement mechanism. Specifically, tasks are assigned processor time *budgets* that are consumed as the tasks execute. When the budgets are exhausted, the tasks become ineligible to execute. Additional rules define when and how budgets are replenished.

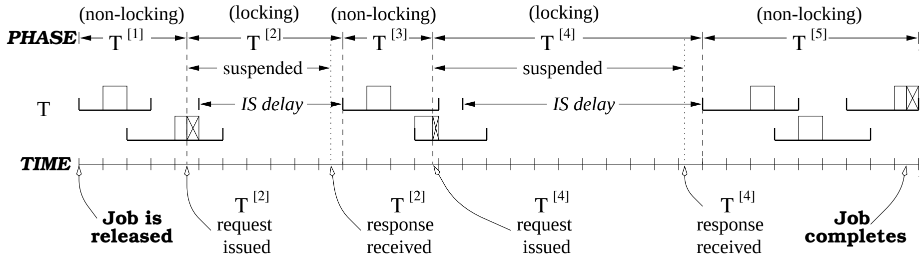


Fig. 8. Illustration of the behavior of a five-phase task T with weight 7/19 when all locking phases use the SWSP. Phases $T^{[1]}$, $T^{[3]}$, and $T^{[5]}$ do not require locks, and hence are executed locally by T while phases $T^{[2]}$ and $T^{[4]}$ are executed remotely by lock servers. Phase transitions are shown across the top of the figure and time is shown across the bottom. Arrows show when each request begins and ends. Unshaded boxes show where T executes locally while boxes containing X's denote unutilized processor time.

Detailed description. Let $V(\ell)$ denote the server task that implements lock ℓ. Each server is assumed to maintain a FIFO-ordered request queue, $\mathsf{WAIT}\,(\ell)$. As long as $\mathsf{WAIT}\,(\ell)$ is non-empty, $V(\ell)$ releases subtasks as early as is permitted under the GIS task model. On the other hand, if $V(\ell)$ 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 a synchronous call with behavior similar to an RPC, *i.e.*, T is suspended until the response is received.

Figure 8 illustrates the behavior of one job of a five-phase task T when all locking phases use the SWSP. As shown, server delays can be treated just as any other form of suspension with one exception: the local processing required to initiate the request and process the response must be factored into the execution-time estimates of the phases that precede and succeed, respectively, the locking phase. (When using the SP and the RP, this overhead can be factored into the locking phase's requirements.)

Analysis. Before continuing, we define additional shorthand expressions unique to the SWSP analysis. First, let $\delta\,(A, w)$ denote 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, where A is a positive integer. By Lemma 2.3,

$$\delta\,(A, w) = \left\lceil \frac{A + \beta - 1}{w} \right\rceil + \epsilon.$$

Informally, $\delta\,(A, w)$ is one slot less than the worst-case span of any $A+1$ consecutive windows, as illustrated in Figure 9.

The following theorem and corollary equate an SWSP locking phase to a suspension phase. Recall that $\mathbf{I}(\theta)$ denotes a suspension phase of maximum duration θ.

Theorem 6.1. *Given a phase $T^{[i]} \in \mathcal{C}(\ell)$, where ℓ is managed by the SWSP, phase $T^{[i]}$ is equivalent to*

$$\mathbf{I}\left(\delta\left(\left\lceil \frac{T^{[i]}.e + I(T, \ell)}{Q} \right\rceil, V(\ell).w\right)\right).$$

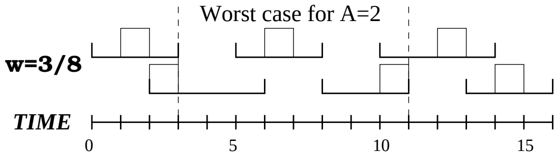


Fig. 9. The marked interval shows the interval that defines $\delta\,(A, w)$ when $w = \frac{3}{8}$, $A = 2$, $\beta = 2$, and $\epsilon = 0$. In this case, $\delta\,(A, w) = \lceil (2 + 2 - 1)\frac{8}{3} \rceil + 0 = 8$. As shown, $\delta\,(A, w)$ is one slot shorter than the longest span of any $A + 1$ consecutive windows.

Proof. Since $I(T, \ell)$ is a trivial upper bound on the time required by competing requests, the server cannot consume more than $\left\lceil \frac{T^{[i]}.e + I(T,\ell)}{Q} \right\rceil$ quanta without completing T's critical section. Since the server remains active while T's request is pending, it follows that the server completes T's critical section after a delay of at most $\delta\left(\left\lceil \frac{T^{[i]}.e+I(T,\ell)}{Q} \right\rceil, V(\ell).w\right)$ slots. $\square$

Assigning server weights. We now describe a simple algorithm for assigning weights to servers. Our algorithm focuses on dividing a fixed fraction of the processor bandwidth reserved for servers, denoted ∂, among those servers in proportion to their requirements. Let

$$\mathcal{U}(T, \ell) = \frac{\sum_{T^{[i]} \in \mathcal{C}(\ell)} T^{[i]}.e}{T.p}.$$

Informally, $\mathcal{U}(T, \ell)$ is T's utilization of lock ℓ. Also, let

$$\mathcal{U}(\ell) = \sum_{T \in \tau} \mathcal{U}(T, \ell),$$

which is the total utilization of lock ℓ by all tasks in τ. (We also refer to this quantity as the *lock utilization* of ℓ.) ∂ can be proportionally distributed among the servers by using lock utilizations like relative weights,[16] as suggested by the formula below.

$$V(\ell).w = \frac{\mathcal{U}(\ell)}{\sum_{\ell' \in \Gamma} \mathcal{U}(\ell')} \cdot \partial$$

7. EXPERIMENTAL RESULTS

In this section, we present a simple experimental comparison of the zone-based and server-based protocols. Specifically, we consider the mean synchronization overhead experienced under each approach by randomly generated task sets for which (R3) holds. Based on Ramamurthy's observations [Ramamurthy 1997], this is expected to be the most common scenario in practice.

[16]In the presented formula, we ignore the unit upper limit on weights. An algorithm for dividing bandwidth while respecting such a restriction can be found in [Chandra et al. 2000].

Synchronization overhead is the difference between the total weight of all tasks under Pfair scheduling and the utilization of the task set τ, which is defined as shown below.

$$\tau.u = \sum_{T \in \tau} \frac{T.e}{T.p}$$

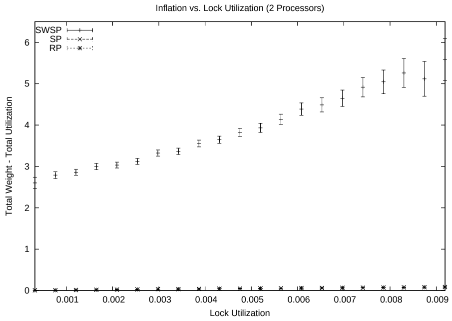
For these experiments, we restrict attention to task sets in which $T.p = T.d$ for all $T \in \tau$. The results presented here are a small fraction of the results from a broader comparison that can be found in [Holman 2004].

Setup. Task sets were generated for each of 2-, 4-, 8-, and 16-processor systems. The task count and total utilization were systematically varied over the ranges $5 \cdot \log_2 M, \ldots, 30 \cdot \log_2 M$ and $0.2 \cdot M, \ldots, 0.8 \cdot M$, respectively. In addition, the lock count was systematically varied over the range $\log_2 M, \ldots, 10 \cdot \log_2 M$. These ranges were all chosen arbitrarily based on ranges that seem likely to occur in real systems. To ensure that (R3) held, critical section durations were upper bounded by 0.025, *i.e.*, 2.5% of the slot length. When generating individual tasks, a maximum phase count of twenty-four was imposed. Also, critical sections occurred in consecutive phases with probability $\frac{1}{4}$. Finally, all $T.p$ values were randomly chosen from the range $50, \ldots, 2000$.

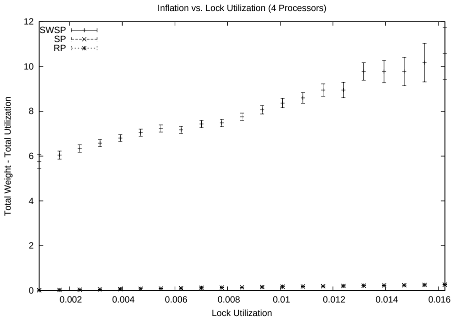
Sampling. To enable a quantitative comparison, only *valid* task sets were considered. A task set must satisfy several constraints to be considered valid. First, under the zone-based protocols, all locks must satisfy (R3) and the task set must be schedulable on M processors. (Since the analysis presented earlier for the SP and the RP is based on M, we must verify that the task set is actually schedulable on M processors for the analysis to be correct.) Second, the task set must be feasible under the SWSP when using an unlimited number of processors. Feasibility can be checked by computing task weights for the special case in which each server is assigned a unit weight. Such a case is guaranteed to produce the lowest possible task weights under the SWSP due to the fact that each server exhibits the best possible responsiveness. Hence, if a task weight exceeds unity under these conditions, then the task set is not schedulable under the SWSP, regardless of the processor count.

Results. The calculated overhead is plotted against the total lock utilization of the task set in Figure 10. In each inset, the sample means are shown for each approach with 99% confidence intervals. As shown, the RP and SP consistently outperform the SWSP, which produces more than an order of magnitude higher overhead. Indeed, the server weights actually account for the bulk of this overhead.

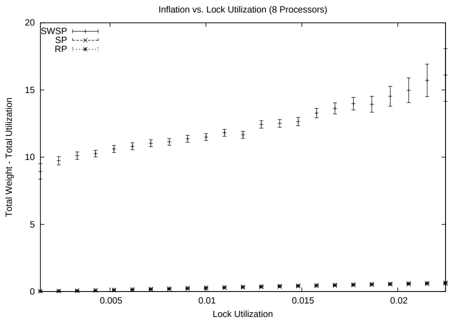
Due to the large scale of the SWSP's overhead, the relative overhead of the SP and RP cannot be determined from Figure 10. Figure 11 omits the SWSP to enable such a comparison. Indeed, these results show exactly what the earlier analysis suggests: the performance is comparable for small values of M, but the SP increasingly outperforms the RP as M is increased. The explanation for this is simple. Recall that the idea underlying the RP is to avoid the need to save and restore request state at slot boundaries by sacrificing the guarantee that a request cannot be blocked by the same competing request multiple times. Since only one competing request is present on each processor per slot, the impact of this



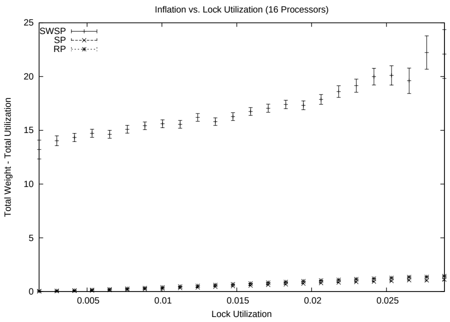
(a) $M = 2$



(b) $M = 4$



(c) $M = 8$



(d) $M = 16$

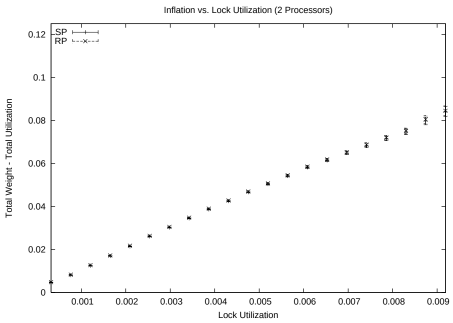
Fig. 10. Plots show synchronization overhead under the SP, RP, and SWSP as a function of lock utilization. The 99% confidence interval is show for each point. The figure shows **(a)** the $M = 2$, **(b)** the $M = 4$, **(c)** the $M = 8$, and **(d)** the $M = 16$ cases.

sacrifice is directly proportional to M. Hence, as M increases, the SP guarantees less synchronization overhead since it maintains that guarantee while the RP does not.

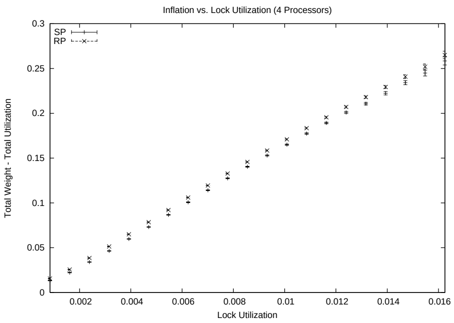
However, it is important to note that the benefit of using the RP instead of the SP is not reflected in this experiment. By not maintaining requests across slots, the RP produces less per-slot overhead than the SP, which will result in more useful processor time per slot (*i.e.*, a larger Q value). Hence, when the performance of the RP and SP is comparable, the RP should be preferred. These issues and others are discussed in much greater detail in [Holman 2004].

8. CONCLUSION

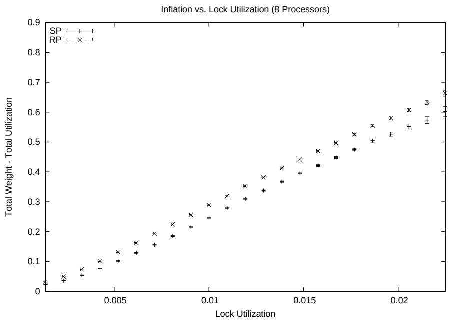
In this paper, we have addressed the problem of providing support for lock-based synchronization in Pfair-scheduled multiprocessor systems. We began by generalizing support for periodic and sporadic tasks under Pfair scheduling. Next, we considered the viability of using inheritance-based protocols. As part of this discussion, we compared several forms of inheritance that are available under Pfair scheduling. After this, we presented protocols, along with supporting analysis, to handle both short (relative to Q) critical sections and critical sections of arbitrary



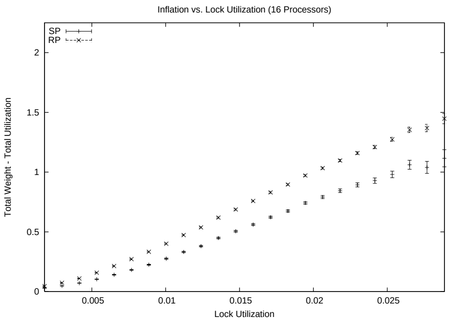
(a) $M = 2$



(b) $M = 4$



(c) $M = 8$



(d) $M = 16$

Fig. 11. Plots show synchronization overhead under the SP and RP as a function of lock utilization. The 99% confidence interval is show for each point. The figure shows **(a)** the $M = 2$, **(b)** the $M = 4$, **(c)** the $M = 8$, and **(d)** the $M = 16$ cases. These plots are close-ups of those shown in Figure 10.

length. For the former case, we demonstrated how the quantum-based nature of Pfair scheduling can be exploited to avoid the preemption of lock-holding tasks. For the latter case, we presented a simple server-based protocol that emphasizes ease-of-use. The protocols presented in this paper have the advantage that they can be applied on a per-lock basis, and hence can be used together in the same system. Finally, we presented a simple experimental evaluation of the synchronization overhead suffered under each presented protocol when (R3) holds.

Due to length considerations, we have not presented all results relating to this work. A more complete coverage of this topic can be found in [Holman 2004]. Additional results not presented here include a more extensive experimental comparison of the proposed protocols, sample pseudo-code implementations of the SP and RP, and optimization techniques that strive to reduce the worst-case synchronization overhead.

APPENDIX

In this appendix, we derive the basic properties of scheduling summarized earlier in Section 2.

Window placement. Theorem 2.1 is established by a trivial extension of the lemma shown below.

Lemma A.1. *The following formulas define the placement of relaxed windows:*

$$r(T_i) = \left\lfloor \frac{i - \beta_+}{T.w} \right\rfloor \qquad\qquad d(T_i) = \left\lceil \frac{(i-1) + \beta_-}{T.w} \right\rceil .$$

Proof. These formulas follow directly from (5), which implies that the release and deadline of a subtask T_i are defined as follows:

$$r(T_i) = \min\{\ k \mid k \in \mathcal{Z} \wedge i \cdot Q - \mathit{fluid}(T, 0, k+1) < Q \cdot \beta_+ \ \};\ \text{and}$$
$$d(T_i) = \min\{\ k \mid k \in \mathcal{Z} \wedge (i-1) \cdot Q - \mathit{fluid}(T, 0, k) \leq -Q \cdot \beta_- \ \}.$$

In the above formulas, $\mathcal{Z}$ denotes the set of all integers. Informally, the $r(T_i)$ constraint identifies the earliest slot (k) such that the upper lag constraint $(Q \cdot \beta_+)$ is not violated when the i^{th} quanta is received in that slot (*i.e.*, in the interval $[k, k+1)$). The subtask release corresponds to the start of this slot, *i.e.*, time k.[17] On the other hand, the $d(T_i)$ constraint identifies the earliest time k such that the lower lag bound $(-Q \cdot \beta_-)$ *is* violated when only $i-1$ quanta are received by k. It follows that the i^{th} quantum must be received in the interval $[k-1, k)$, at the latest. Hence, the subtask deadline occurs at time k.

Applying (2) and rearranging terms to isolate k produces the following equivalent forms.

$$r(T_i) = \min\left\{ k \;\middle|\; k \in \mathcal{Z} \wedge k > \frac{i - \beta_+}{T.w} - 1 \right\}$$
$$d(T_i) = \min\left\{ k \;\middle|\; k \in \mathcal{Z} \wedge k \geq \frac{(i-1) + \beta_-}{T.w} \right\}$$

The lemma follows. □

Window overlap. Lemma A.2, shown below, bounds the number of slots by which consecutive subtask windows can overlap in the absence of IS delays.

Lemma A.2. *Each pair of consecutive subtasks T_i and T_{i+1} satisfy the inequality shown below.*

$$\left\lceil \frac{\beta - 2}{T.w} \right\rceil + \epsilon \leq d(T_i) - r(T_{i+1}) \leq \left\lceil \frac{\beta - 2}{T.w} \right\rceil + \epsilon + 1$$

If T_i is the last subtask in a cycle, then

$$d(T_i) - r(T_{i+1}) = \mathcal{B}(T) \stackrel{\text{def}}{=} \left\lceil \frac{\beta_- - 1}{T.w} \right\rceil + \left\lceil \frac{\beta_+ - 1}{T.w} \right\rceil + \epsilon.$$

Proof. The following derivation establishes the upper bound.

[17] Notice that the release time may be negative. It is important to understand that time 0 is simply a reference point that records when scheduling begins. Since no scheduling occurs prior to this point, windows with negative release times are effectively truncated to begin at time 0.

$$
\begin{aligned}
&d(T_i) - r(T_{i+1}) \\
&= \left(\left\lceil \tfrac{(i-1)+\beta_-}{T.w} \right\rceil + \epsilon_d\right) - \left(\left\lfloor \tfrac{(i+1)-\beta_+}{T.w} \right\rfloor - \epsilon_r\right) && \text{, by Theorem 2.1} \\
&\le \left\lceil \tfrac{\beta_+ + \beta_- - 2}{T.w} \right\rceil + \left\lfloor \tfrac{i+1-\beta_+}{T.w} \right\rfloor + 1 - \left\lfloor \tfrac{i+1-\beta_+}{T.w} \right\rfloor + \epsilon_r + \epsilon_d, && \lceil a+b \rceil \le \lceil a \rceil + \lfloor b \rfloor + 1 \\
&= \left\lceil \tfrac{\beta_+ + \beta_- - 2}{T.w} \right\rceil + 1 + \epsilon_r + \epsilon_d && \text{, simplification} \\
&= \left\lceil \tfrac{\beta - 2}{T.w} \right\rceil + \epsilon + 1 && \text{, by (6) and (7)}
\end{aligned}
$$

The following derivation establishes the lower bound.

$$
\begin{aligned}
&d(T_i) - r(T_{i+1}) \\
&= \left(\left\lceil \tfrac{(i-1)+\beta_-}{T.w} \right\rceil + \epsilon_d\right) - \left(\left\lfloor \tfrac{(i+1)-\beta_+}{T.w} \right\rfloor - \epsilon_r\right) && \text{, by Theorem 2.1} \\
&\ge \left\lceil \tfrac{\beta_+ + \beta_- - 2}{T.w} \right\rceil + \left\lfloor \tfrac{i+1-\beta_+}{T.w} \right\rfloor - \left\lfloor \tfrac{i+1-\beta_+}{T.w} \right\rfloor + \epsilon_r + \epsilon_d && \text{, } \lceil a+b \rceil \ge \lceil a \rceil + \lfloor b \rfloor \\
&= \left\lceil \tfrac{\beta_+ + \beta_- - 2}{T.w} \right\rceil + \epsilon_r + \epsilon_d && \text{, simplification} \\
&= \left\lceil \tfrac{\beta - 2}{T.w} \right\rceil + \epsilon && \text{, by (6) and (7)}
\end{aligned}
$$

When T_i is the last subtask in a cycle, then $\frac{i}{T.w}$ is an integer since $\mathcal{E}(T) \mid i$ holds. This leads to the derivation shown below.

$$
\begin{aligned}
&d(T_i) - r(T_{i+1}) \\
&= \left(\left\lceil \tfrac{(i-1)+\beta_-}{T.w} \right\rceil + \epsilon_d\right) - \left(\left\lfloor \tfrac{(i+1)-\beta_+}{T.w} \right\rfloor - \epsilon_r\right) && \text{, by Theorem 2.1} \\
&= \tfrac{i}{T.w} + \left\lceil \tfrac{\beta_- - 1}{T.w} \right\rceil - \tfrac{i}{T.w} - \left\lfloor \tfrac{1-\beta_+}{T.w} \right\rfloor + \epsilon_r + \epsilon_d && \text{, since } \tfrac{i}{T.w} \text{ is an integer} \\
&= \left\lceil \tfrac{\beta_- - 1}{T.w} \right\rceil - \left\lfloor \tfrac{1-\beta_+}{T.w} \right\rfloor + \epsilon_r + \epsilon_d && \text{, simplification} \\
&= \left\lceil \tfrac{\beta_- - 1}{T.w} \right\rceil + \left\lceil \tfrac{\beta_+ - 1}{T.w} \right\rceil + \epsilon_r + \epsilon_d && \text{, } \lfloor -a \rfloor = -\lceil a \rceil \\
&= \left\lceil \tfrac{\beta_- - 1}{T.w} \right\rceil + \left\lceil \tfrac{\beta_+ - 1}{T.w} \right\rceil + \epsilon && \text{, by (7)}
\end{aligned}
$$

The above derivations establish the lemma. □

Window span. Lemma A.3, shown below, bounds the number of slots spanned by a sequence of n consecutive windows.

LEMMA A.3. *Every sequence of consecutive subtasks $T_{i+1}, \ldots, T_{i+n}$ satisfies the following:*

$$\left\lceil \frac{n+\beta-2}{T.w} \right\rceil + \epsilon \le d(T_{i+n}) - r(T_{i+1}) \le \left\lceil \frac{n+\beta-2}{T.w} \right\rceil + \epsilon + 1$$

When T_{i+1} begins a cycle (i.e., $\mathcal{E}(T) \mid i$ holds) and subtask $i+n$ ends a cycle (i.e., $\mathcal{E}(T) \mid (i+n)$ holds),

$$d(T_{i+n}) - r(T_{i+1}) = \frac{n}{T.w} + \mathcal{B}(T)\,.$$

PROOF. The following derivation establishes the upper bound in the first claim.

$$
\begin{aligned}
&d(T_{i+n}) - r(T_{i+1}) \\
&= \left(\left\lceil \tfrac{(i+n-1)+\beta_-}{T.w} \right\rceil + \epsilon_d\right) - \left(\left\lfloor \tfrac{(i+1)-\beta_+}{T.w} \right\rfloor - \epsilon_r\right) && \text{, by Theorem 2.1} \\
&\leq \left\lceil \tfrac{n+\beta_+ + \beta_- - 2}{T.w} \right\rceil + \left\lfloor \tfrac{i+1-\beta_+}{T.w} \right\rfloor + 1 - \left\lfloor \tfrac{i+1-\beta_+}{T.w} \right\rfloor + \epsilon_r + \epsilon_d \\
& && \text{, } \lceil a+b \rceil \leq \lceil a \rceil + \lfloor b \rfloor + 1 \\
&= \left\lceil \tfrac{n+\beta_+ + \beta_- - 2}{T.w} \right\rceil + \epsilon_r + \epsilon_d + 1 && \text{, simplification} \\
&= \left\lceil \tfrac{n+\beta-2}{T.w} \right\rceil + \epsilon + 1 && \text{, by (6) and (7)}
\end{aligned}
$$

The following derivation establishes the lower bound in the first claim.

$$
\begin{aligned}
&d(T_{i+n}) - r(T_{i+1}) \\
&= \left(\left\lceil \tfrac{(i+n-1)+\beta_-}{T.w} \right\rceil + \epsilon_d\right) - \left(\left\lfloor \tfrac{(i+1)-\beta_+}{T.w} \right\rfloor - \epsilon_r\right) && \text{, by Theorem 2.1} \\
&\geq \left\lceil \tfrac{n+\beta_+ + \beta_- - 2}{T.w} \right\rceil + \left\lfloor \tfrac{i+1-\beta_+}{T.w} \right\rfloor - \left\lfloor \tfrac{i+1-\beta_+}{T.w} \right\rfloor + \epsilon_r + \epsilon_d && \text{, } \lceil a+b \rceil \geq \lceil a \rceil + \lfloor b \rfloor \\
&= \left\lceil \tfrac{n+\beta_+ + \beta_- - 2}{T.w} \right\rceil + \epsilon_r + \epsilon_d && \text{, simplification} \\
&= \left\lceil \tfrac{n+\beta-2}{T.w} \right\rceil + \epsilon && \text{, by (6) and (7)}
\end{aligned}
$$

When $\mathcal{E}(T) \mid i$ and $\mathcal{E}(T) \mid (i+n)$ both hold, it follows that both $\frac{i}{T.w}$ and $\frac{i+n}{T.w}$ are integers. Hence, the derivation shown below establishes the second claim.

$$
\begin{aligned}
&d(T_{i+n}) - r(T_{i+1}) \\
&= \left(\left\lceil \tfrac{(i+n-1)+\beta_-}{T.w} \right\rceil + \epsilon_d\right) - \left(\left\lfloor \tfrac{(i+1)-\beta_+}{T.w} \right\rfloor - \epsilon_r\right) && \text{, by Theorem 2.1} \\
&= \tfrac{i+n}{T.w} + \left\lceil \tfrac{\beta_- - 1}{T.w} \right\rceil - \tfrac{i}{T.w} - \left\lfloor \tfrac{1-\beta_+}{T.w} \right\rfloor + \epsilon_r + \epsilon_d && \text{, since } \tfrac{i}{T.w} \text{ and } \tfrac{i+n}{T.w} \text{ are integers} \\
&= \tfrac{n}{T.w} + \left\lceil \tfrac{\beta_- - 1}{T.w} \right\rceil - \left\lfloor \tfrac{1-\beta_+}{T.w} \right\rfloor + \epsilon_r + \epsilon_d && \text{, simplification} \\
&= \tfrac{n}{T.w} + \left\lceil \tfrac{\beta_- - 1}{T.w} \right\rceil + \left\lceil \tfrac{\beta_+ - 1}{T.w} \right\rceil + \epsilon_r + \epsilon_d && \text{, } \lfloor -a \rfloor = -\lceil a \rceil \\
&= \tfrac{n}{T.w} + \left\lceil \tfrac{\beta_- - 1}{T.w} \right\rceil + \left\lceil \tfrac{\beta_+ - 1}{T.w} \right\rceil + \epsilon && \text{, by (7)} \\
&= \tfrac{n}{T.w} + \mathcal{B}(T) && \text{, by Lemma A.2}
\end{aligned}
$$

This completes the proof. □

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their helpful suggestions regarding an earlier draft of this paper.

REFERENCES

Anderson, J., Block, A., and Srinivasan, A. 2003. Quick-release fair scheduling. In *Proceedings of the 24th IEEE Real-time Systems Symposium*. 130–141.

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

Anderson, J. and Srinivasan, A. 2000a. Early-release fair scheduling. In *Proceedings of the 12th Euromicro Conference on Real-time Systems*. 35–43.

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

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

BAKER, T. 1991. Stack-based scheduling of real-time processes. *Real-time Systems 3,* 1 (Mar.), 67–99.

BARUAH, S., COHEN, N., PLAXTON, C., AND VARVEL, D. 1996. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica 15*, 600–625.

BARUAH, S., GEHRKE, J., AND PLAXTON, C. G. 1995. Fast scheduling of periodic tasks on multiple resources. In *Proceedings of the 9th International Parallel Processing Symposium.* 280–288.

BLOCK, A., ANDERSON, J., AND BISHOP, G. 2005. Fine-grained task reweighting on multiprocessors. In *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (to appear).*

CACCAMO, M. AND SHA, L. 2001. Aperiodic servers with resource constraints. In *Proceedings of the 22nd IEEE Real-time Systems Symposium.* 161–170.

CHANDRA, A., ADLER, M., GOYAL, P., AND SHENOY, P. 2000. Surplus fair scheduling: A proportional-share cpu scheduling algorithm for symmetric multiprocessors. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation.*

DE NIZ, D., ABENI, L., SAEWONG, S., AND RAJKUMAR, R. 2001. Resource sharing in reservation-based systems. In *Proceedings of the 22nd IEEE Real-time Systems Symposium.* 171–180.

GAI, P., LIPARI, G., AND DI NATALE, M. 2001. 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.* 73–83.

HOLMAN, P. 2004. On the implementation of Pfair-scheduled multiprocessor systems. Ph.D. thesis, University of North Carolina at Chapel Hill.

HOLMAN, P. AND ANDERSON, J. 2002a. Locking in Pfair-scheduled multiprocessor systems. In *Proceedings of the 23rd IEEE Real-time Systems Symposium.* 149–158.

HOLMAN, P. AND ANDERSON, J. 2002b. Object sharing in Pfair-scheduled multiprocessor systems. In *Proceedings of the 14th Euromicro Conference on Real-time Systems.* 111–120.

HOLMAN, P. AND ANDERSON, J. 2003. Using hierarchal scheduling to improve resource utilization in multiprocessor real-time systems. In *Proceedings of the 15th Euromicro Conference on Real-time Systems.* 41–50.

HOLMAN, P. AND ANDERSON, J. 2005. Supporting lock-free synchronization in Pfair-scheduled systems. *Journal of Parallel and Distributed Computing (to appear).*

LAMASTRA, G., LIPARI, G., AND ABENI, L. 2001. A bandwidth inheritance algorithm for real-time task synchronization in open systems. In *Proceedings of the 22nd IEEE Real-time Systems Symposium.* 151–160.

LIU, C. AND LAYLAND, J. 1973. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM 30*, 46–61.

LYNX REAL-TIME SYSTEMS. 1993. *LynxOS application writer's guide.* Lynx Real-time Systems, Inc.

MOIR, M. AND RAMAMURTHY, S. 1999. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In *Proceedings of the Twentieth IEEE Real-time Systems Symposium.* 294–303.

MOK, A. 1983. Fundamental design problems for the hard real-time environment. Ph.D. thesis, Massachussetts Institute of Technology.

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

RAJKUMAR, R. 1991. *Synchronization in real-time systems – A priority inheritance approach.* Kluwer Academic Publishers, Boston, Mass.

RAJKUMAR, R., SHA, L., AND LEHOCZKY, J. 1988. Real-time synchronization protocols for multiprocessors. In *Proceedings of the Ninth IEEE Real-time Systems Symposium.* 259–269.

RAMAMURTHY, S. 1997. A lock-free approach to object sharing in real-time systems. Ph.D. thesis, University of North Carolina at Chapel Hill.

SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. 1990. Priority inheritance protocols: An approach to real-time system synchronization. *IEEE Transactions on Computers 39,* 9, 1175–1185.

Srinivasan, A. and Anderson, J. 2002. Optimal rate-based scheduling on multiprocessors. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*. 189–198.

Srinivasan, A. and Anderson, J. 2003a. Efficient scheduling of soft real-time applications on multiprocessors. In *Proceedings of the 15th Euromicro Conference on Real-time Systems*. 51–59.

Srinivasan, A. and Anderson, J. 2003b. Fair scheduling of dynamic task systems on multiprocessors. In *11th International Workshop on Parallel and Distributed Real-time Systems (on CD-ROM)*.