Fair Scheduling of Real-time Tasks on Multiprocessors

James Anderson, Philip Holman, and Anand Srinivasan

Department of Computer Science, University of North Carolina at Chapel Hill

1 Introduction

There has been much recent interest in fair scheduling algorithms for real-time multiprocessor systems. The roots of much of the research on this topic can be traced back to the seminal work of Baruah *et al.* on *Proportionate fairness* (*Pfairness*) [6]. This work proved that the problem of optimally scheduling periodic tasks[1] on multiprocessors could be solved *on-line* in polynomial time by using Pfair scheduling algorithms. Pfair scheduling differs from more conventional real-time scheduling approaches in that tasks are explicitly required to execute at steady rates. In most real-time scheduling disciplines, the notion of a rate is implicit. For example, in a periodic schedule, a task T executes at a rate defined by its required utilization ($T.e/T.p$) over large intervals. However, T's execution rate over short intervals, *e.g.*, individual periods, may vary significantly. Hence, the notion of a rate under the periodic task model is a bit inexact.

Under Pfair scheduling, each task is executed at an approximately uniform rate by breaking it into a series of quantum-length *subtasks*. Time is then subdivided into a sequence of (potentially overlapping) subintervals of approximately equal lengths, called *windows*. To satisfy the Pfairness

[1] A periodic task T is characterized by a *phase* $T.\phi$, an *execution requirement* $T.e$, and a *period* $T.p$: a job release (*i.e.*, task invocation) occurs at time $T.\phi + (k-1) \cdot T.p$ for each integer $k \geq 1$ and the k^{th} job must receive $T.e$ units of processor time by the next release (at time $T.\phi + k \cdot T.p$). A periodic task system is *synchronous* if each task in the system has a phase of 0.

rate constraint, each subtask must execute within its associated window. Different subtasks of a task are allowed to execute on different processors (*i.e.*, inter-processor migration is permitted), but may not execute simultaneously (*i.e.*, parallelism is prohibited).

By breaking tasks into uniform-sized subtasks, Pfair scheduling circumvents many of the bin-packing-like problems that lie at the heart of intractability results that pertain to multiprocessor scheduling. Indeed, Pfair scheduling is presently the only known approach for optimally scheduling periodic tasks on multiprocessors. Three Pfair scheduling algorithms have been proven optimal: PF [6], PD [7], and PD^2 [2]. Several suboptimal algorithms have also been proposed, including the earliest-pseudo-deadline-first (EPDF) algorithm [16, 21], the weight-monotonic (WM) algorithm [5, 17], and the deadline-fair-scheduling (DFS) algorithm [9].

In this chapter, we present an overview of Pfair scheduling and many of the extensions to it that have been proposed. In Section 2, we formally describe Pfairness and its relationship to periodic task scheduling. In Section 3, we present the *intra-sporadic fairness* (ISfairness) [3, 20] constraint, which extends the Pfairness constraint to systems in which the execution of subtasks may be delayed. In Section 4, we present results relating to dynamic task systems, *i.e.*, systems in which tasks are allowed to leave and join [22]. Section 5 describes prior work on using hierarchical scheduling under a global Pfair scheduler [11, 16]. We then present resource-sharing protocols [12, 13] designed for Pfair-scheduled systems in Section 6. Section 7 summarizes the chapter.

2 Periodic Task Systems

In this section, we consider the problem of scheduling a set τ of synchronous periodic tasks on a multiprocessor. We assume that processor time is allocated in discrete time units, or *quanta*, and refer to the time interval $[t, t+1)$, where t is a nonnegative integer, as *slot* t. We further assume that all task parameters are expressed as integer multiples of the quantum size. As mentioned in the footnote in Section 1, each periodic task T is characterized by a *period* $T.p$, a per-job *execution requirement* $T.e$, and a *phase* $T.\phi$. (Recall that $T.\phi = 0$ for synchronous tasks.) We refer to the ratio of $T.e/T.p$ as the *weight* of task T, denoted $wt(T)$. Informally, $wt(T)$ is the rate at which T should be executed, relative to the speed of a single processor. A task with weight less than 1/2 is called a *light* task, while a task with weight at least 1/2 is called a *heavy* task.

The sequence of scheduling decisions over time defines a *schedule*. Formally, a schedule S is a mapping $S : \tau \times \mathcal{Z} \mapsto \{0, 1\}$, where τ is a set of periodic tasks and $\mathcal{Z}$ is the set of nonnegative integers. If $S(T, t) = 1$, then we say that *task T is scheduled in slot t*.

2.1 Pfair Scheduling

In a perfectly fair (ideal) schedule for a synchronous task system, every task T would receive $wt(T) \cdot t$ quanta over the interval $[0, t)$ (which implies that all deadlines are met). However, such idealized sharing is not possible in a quantum-based schedule. Instead, Pfair scheduling algorithms strive to "closely track" the allocation of processor time in the ideal schedule. This tracking is formalized in the notion of per-task *lag*, which is the difference between each task's allocation in the Pfair schedule and the allocation it would receive in an ideal schedule. Formally, the *lag of task T at time t*, denoted $lag(T, t)$,[2] is defined as follows:

$$lag(T, t) = wt(T) \cdot t - \sum_{u=0}^{t-1} S(T, u). \qquad (1)$$

[2]For brevity, we leave the schedule implicit and use $lag(T, t)$ instead of $lag(T, t, S)$.

A schedule is *Pfair* if and only if

$$(\forall T, t :: -1 < lag(T, t) < 1). \tag{2}$$

Informally, (2) requires each task T's allocation error to be less than one quantum at all times, which implies that T must receive either $\lfloor wt(T) \cdot t \rfloor$ or $\lceil wt(T) \cdot t \rceil$ quanta by time t.

It is straightforward to show that the Pfairness constraint subsumes the periodic scheduling constraint. For example, in the case of a synchronous system, a periodic task T must be granted $T.e$ quanta in each interval $[k \cdot T.p, (k+1) \cdot T.p)$, where $k \geq 0$. At $t = k \cdot T.p$, $wt(T) \cdot t = (T.e/T.p) \cdot (k \cdot T.p) = k \cdot T.e$, which is an integer. By (2), each task's allocation in a Pfair schedule will match that of the ideal schedule at period boundaries. Since all deadlines are met in the ideal schedule, they must also be met in each possible Pfair schedule.

Windows. Under Pfair scheduling, each task T is effectively divided into an infinite sequence of quantum-length *subtasks*. We denote the i^{th} subtask of task T as T_i, where $i \geq 1$. The lag bounds in (2) constrain each subtask T_i to execute in an associated *window*, denoted $w(T_i)$. $w(T_i)$ extends from T_i's *pseudo-release*, denoted $r(T_i)$, to its *pseudo-deadline*, denoted $d(T_i)$. (For brevity, we often drop the "pseudo-" prefix.) $r(T_i)$ and $d(T_i)$ are formally defined as shown below.

$$r(T_i) = \left\lfloor \frac{i-1}{wt(T)} \right\rfloor \tag{3}$$

$$d(T_i) = \left\lceil \frac{i}{wt(T)} \right\rceil \tag{4}$$

As an example, consider a task T with weight $wt(T) = 8/11$. Each job of this task consists of eight windows, one for each of its subtasks. Using Equations (3) and (4), it is easy to show that the

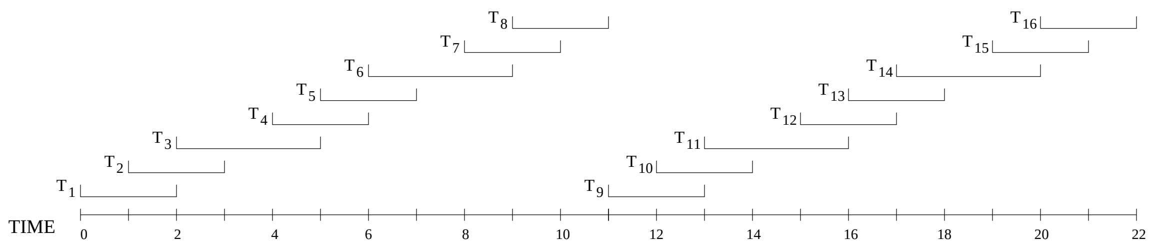


Figure 1: The Pfair windows of the first two jobs (or sixteen subtasks) of a synchronous periodic task T with $T.e = 8$ and $T.p = 11$. Each subtask must be scheduled within its window in order to satisfy (2).

windows within each job of T are as depicted in Figure 1.

Feasibility. A Pfair schedule on M processors exists for τ if and only if

$$\sum_{T \in \tau} \frac{T.e}{T.p} \leq M. \tag{5}$$

This result was proved by Baruah *et al.* by means of a network-flow construction [6]. Let L denote the least common multiple of $\{T.p \mid T \in \tau\}$. By restricting attention to subtasks that fall within the first hyperperiod of τ (*i.e.*, in the interval $[0, L)$), the sufficiency of (5) can be established by constructing a flow graph with integral edge capacities and by then applying the Ford-Fulkerson result [10] to prove the existence of an integer-valued maximum flow for that graph. This integral flow defines a correct schedule over $[0, L)$. (Interested readers are referred to [6] for a detailed proof of this result).

2.2 Optimal Pfair Algorithms

At present, three Pfair scheduling algorithms have been proven optimal for scheduling synchronous periodic tasks on multiprocessors: PF [6], PD [7], and PD^2 [2]. These algorithms prioritize subtasks on an earliest-pseudo-deadline-first (EPDF) basis, but differ in the choice of tie-breaking rules. Breaking ties appropriately turns out to be the key concern when designing optimal Pfair algorithms. One tie-break parameter that is common to all three algorithms is the *successor bit*, which is defined as follows:

$$b(T_i) = \left\lceil \frac{i}{wt(T)} \right\rceil - \left\lfloor \frac{i-1}{wt(T)} \right\rfloor. \tag{6}$$

Informally, $b(T_i)$ denotes the number of slots by which T_i's window overlaps T_{i+1}'s window (see (3) and (4)). For example, in Figure 1, $b(T_i) = 1$ for $1 \le i \le 7$ and $b(T_8) = 0$. (Note that the last subtask of a job of a periodic task has a successor bit of 0.) We now briefly describe each of the three algorithms.

The PF algorithm. Under the PF algorithm, subtasks are prioritized as follows: at time t, if subtasks T_i and U_j are both ready to execute, then T_i has higher priority than U_j, denoted $T_i \succ U_j$, if one of the following holds:

(i) $d(T_i) < d(U_j)$.

(ii) $d(T_i) = d(U_j)$ and $b(T_i) > b(U_j)$.

(iii) $d(T_i) = d(U_j)$, $b(T_i) = b(U_j) = 1$, and $T_{i+1} \succ U_{j+1}$.

If neither subtask has priority over the other, then the tie can be broken arbitrarily. Given the PF priority definition, the description of the PF algorithm is simple: at the start of each slot, the M highest priority subtasks (if that many eligible subtasks exist) are selected to execute in that slot.

As shown in Rule (ii), when comparing two subtasks with equal pseudo-deadlines, PF favors a subtask T_i with $b(T_i) = 1$, *i.e.*, if its window overlaps that of its successor. The intuition behind this rule is that executing T_i early prevents it from being scheduled in its last slot. Avoiding the latter possibility is important because it effectively shortens $w(T_{i+1})$, which makes it more difficult to schedule T_{i+1} by its pseudo-deadline. If two subtasks have equal pseudo-deadlines and successor bits of 1, then according to Rule (iii), their successor subtasks are recursively checked. This recursion will halt within $min(T.e, U.e)$ steps, because the last subtask of each job has a successor bit of 0.

In [6], PF was proven optimal using an inductive swapping argument. The crux of the argument is to show that, if there exists a Pfair schedule S such that all decisions in S before slot t are in accordance with PF priorities, then there exists a Pfair schedule S' such that all the scheduling decisions in S' before slot $t+1$ are in accordance with PF priorities. To prove the existence of S', the scheduling decisions in slot t of S are systematically changed so that they respect the PF priority rules, while maintaining the correctness of the schedule. We briefly summarize the swapping arguments used to transform S.

Suppose that T_i and U_j are both eligible to execute in slot t and $T_i \succ U_j$. Furthermore, suppose that, contrary to the PF priority rules, U_j is scheduled in slot t in S, while T_i is scheduled in a later slot t' in S. There are three possibilities.

- $T_i \succ U_j$ **by Rule (i).** Because T_i's pseudo-deadline is less than U_j's pseudo-deadline, and because the windows of consecutive subtasks overlap by at most one slot, U_{j+1} is scheduled at a later slot than T_i. Therefore, T_i and U_j can be directly swapped, as shown in Figure 2(a).

- $T_i \succ U_j$ **by Rule (ii).** By Rule (ii), $b(U_j) = 0$, which implies that U_{j+1}'s window does not overlap that of U_j. Hence, T_i and U_j can be directly swapped without affecting the scheduling of U_{j+1}, as shown in Figure 2(b).

- $T_i \succ U_j$ **by Rule (iii).** In this case, it may not be possible to directly swap T_i and U_j because U_{j+1} may be scheduled in the same slot as T_i (*i.e.*, swapping T_i and U_j would result in U_j and U_{j+1} being scheduled in the same slot). If U_{j+1} is indeed scheduled in slot t', then it is necessary to first swap T_{i+1} and U_{j+1}, as shown in Figure 2(c), which may in turn necessitate

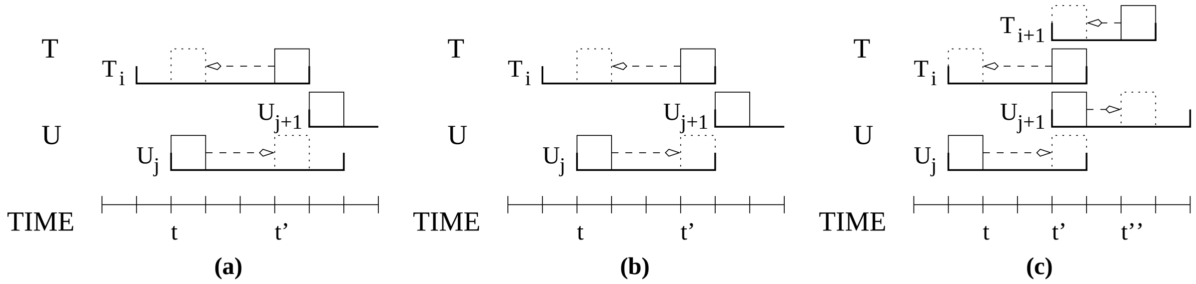


Figure 2: Correctness proof for PF. Dashed arrows indicates the movement of quanta when swapping allocations. Insets (a)–(c) illustrate the three cases considered in the proof.

the swapping of later subtasks.

From this inductive proof, we obtain the following result:

Theorem 1 ([6]) PF *is optimal for scheduling periodic tasks on multiprocessors.*

The PD and PD^2 algorithms. Though optimal, PF is inefficient due to the recursion in Rule (iii). In PD, Rule (iii) is replaced by three additional rules, each of which involves only a simple calculation. (Hence, PD uses four rules to break pseudo-deadline ties.) On the other hand, PD^2 replaces Rule (iii) with a single rule and requires only two tie-break parameters. Since PD^2 is a simplified version of PD, we discuss PD^2 first.

Under Rule (iii) of the PD^2 priority definition, *group deadlines* of competing subtasks are compared. The group deadline of a task T is only important when the task is heavy but does not have unit weight, *i.e.*, when $1/2 \leq wt(T) < 1$. If a task does not satisfy this criterion, then its group deadline is 0.

To motivate the definition of the group deadline, consider a sequence $T_i, \ldots, T_j$ of subtasks such that $b(T_k) = 1 \wedge |w(T_{k+1})| = 2$ for all $i \leq k < j$. Note that scheduling T_i in its last slot forces the other subtasks in this sequence to be scheduled in their last slots. For example, in Figure 1, scheduling T_3 in slot 4 forces T_4 and T_5 to be scheduled in slots 5 and 6, respectively. The group deadline of a subtask T_i, denoted $D(T_i)$, is the earliest time by which such a "cascade" must end. Formally, it is the earliest time t, where $t \geq d(T_i)$, such that either $(t = d(T_k) \;\wedge\; b(T_k) = 0)$ or $(t + 1 = d(T_k) \;\wedge\; |w(T_k)| = 3)$ for some subtask T_k. (Intuitively, if we imagine a job of T in which each subtask is scheduled in the first slot of its window, then the slots that remain empty exactly correspond to the group deadlines of T within that job.) For example, in Figure 1, $D(T_3) = d(T_6) - 1 = 8$ and $D(T_7) = d(T_8) = 11$.

Using this definition, Rule (iii) of the PD^2 priority rules is defined as follows.

(iii) $d(T_i) = d(U_j)$, $b(T_i) = b(U_j) = 1$, and $D(T_i) > D(U_j)$.

(Rules (i) and (ii) remain unchanged, and ties not resolved by all three rules can still be broken arbitrarily.) As shown, PD^2 favors subtasks with later group deadlines. The intuition behind this prioritization is that scheduling these subtasks early prevents (or at least reduces the extent of) cascades. Cascades are undesirable since they constrain the scheduling of future slots.

The priority definition used in the PD algorithm adds two additional tie-break parameters (and rules) to those used by PD^2. The first of these is a task's weight. The second is a bit that distinguishes between the two different types of group deadline. For example, in Figure 1, $D(T_1)$ is a type-1 group deadline because its placement is determined by the $(t + 1 = d(T_k) \;\wedge\; |w(T_k)| = 3)$ condition, while $D(T_6)$ is is a type-0 group deadline. (Effectively, the value of this bit is the value of $b(T_k)$, where T_k is the subtask that defines the group deadline.) The optimality of PD^2 shows that these two additional tie-break parameters are not needed.

PD was proved optimal by a simulation argument that showed that PD "closely tracks" the behavior of the PF algorithm; the optimality of PF was then used to infer the optimality of PD [7]. On the other hand, PD^2 was proved optimal through a swapping technique, similar to that used in optimality proof of PF [2]. We omit these proofs because they are quite lengthy.

Theorem 2 ([7]) PD *is optimal for scheduling periodic tasks on multiprocessors.*

Theorem 3 ([2]) PD^2 *is optimal for scheduling periodic tasks on multiprocessors.*

Implementation. We now describe an implementation strategy that results in a scheduler with $O(M \log N)$ time complexity, where M is the number of processors, and N the number of tasks [7]. First, assume that eligible subtasks are stored in a priority-ordered "ready queue" R and that ineligible subtasks that will become eligible at time t are stored in the "release queue" Q_t. At the beginning of slot t, Q_t is merged into R. The $\min(M, |R|)$ highest-priority subtasks in R are then extracted from R and selected for execution. For each selected subtask T_i, its successor, T_{i+1}, is initialized and inserted into the appropriate release queue. Using binomial heaps (which are capable of performing all basic heap operations in $O(\log N)$ time) to implement the various queues yields the desired $O(M \log N)$ time complexity.

2.3 ERfair Scheduling

One undesirable characteristic of Pfair scheduling is that jobs can be ineligible according to the Pfairness constraint, despite being ready. Consequently, processors may idle while ready but unscheduled jobs exist. *Early-release fairness* (ERfairness) [2] was proposed to address this problem. Under ERfairness, the -1 lag constraint is dropped from (2). Instead, each subtask T_i is assumed to have an *eligibility time* $e(T_i) \leq r(T_i)$, which is the time at which T_i becomes eligible to execute. (The scheduling of T_i must still respect precedence constraints, *i.e.*, T_i cannot become eligible until T_{i-1} is scheduled, regardless of $e(T_i)$.) In [2], PD^2 was shown to correctly schedule any ERfair task set satisfying (5). Note that a Pfair schedule will always respect ERfairness, but not vice versa.

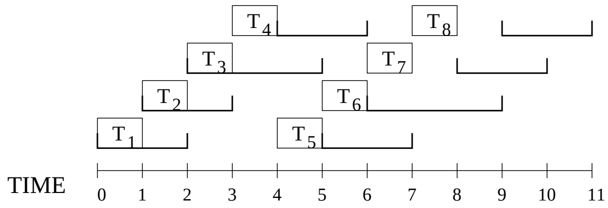


Figure 3: The Pfair windows of the first jobs of a task T with weight 8/11 are shown system. The schedule shown is a valid ERfair schedule, but not a valid Pfair schedule.

Figure 3 illustrates how one job of the task in Figure 1 might be scheduled under ERfair scheduling.

Servicing aperiodic jobs. One important application of ERfair scheduling is to permit mixed scheduling of Pfair tasks and aperiodic jobs. An aperiodic job is a one-shot job that is not associated with any task. Such jobs typically represent the execution of service routines, including those used for interrupt handling. The response times of aperiodic jobs can be improved by allowing server tasks to early-release their subtasks [23]. This improves responsiveness while also ensuring that the schedulability of periodic tasks is not compromised.

Implementation. PF, PD and PD^2 can be easily adapted to allow early releases. In particular, if T_i is selected for execution, and if its successor T_{i+1} is eligible (perhaps due to an early release), then T_{i+1} can be inserted immediately into the ready queue. Hence, ERfair variants tend to be more efficient than their counterparts since fewer queue-merge operations are needed.

2.4 Practicality

Because of the quantum-based nature of Pfair scheduling, the frequency of preemptions and migrations is a potential concern. A recent experimental comparison conducted by Baruah and us [24] showed that PD^2 has comparable performance (in terms of schedulability) to a task-partitioning approach in which each task is statically assigned to a processor and the well-known earliest-deadline-first (EDF) scheduling algorithm is used on each processor. In this study, real system overheads such as context-switching costs were considered. Moreover, the study was biased against Pfair scheduling in that only static systems with independent[3] tasks of low weight[4] were considered. In spite of the frequent context-switching and cache-related overheads, PD^2 performs competitively because the schedulability loss due to these overheads is offset by the fact that PD^2 provides much better analytical bounds than partitioning.

To improve the practicality of Pfair scheduling, Holman and Anderson investigated many techniques for reducing overhead and improving performance in Pfair-scheduled systems [11, 12, 13, 14, 15]. These techniques have targeted many aspects of the system, including the frequency of context switching and migrations, scheduling overhead, cache performance, and contention for shared hardware and software resources.

Chandra, Adler, and Shenoy investigated the use of Pfair scheduling in general-purpose operating systems [9]. The goal of their work was to determine the efficacy of using Pfair scheduling to provide quality-of-service guarantees to multimedia applications. Consequently, no formal analysis of their approach was presented. Despite this, the experimental evaluation of their work convincingly demonstrates the practicality of Pfair scheduling.

[3]Considering only independent tasks is advantageous to EDF. While the synchronization techniques described later in Section 6 permit efficient sharing of global resources, no efficient global techniques have been proposed for partitioned EDF-scheduled systems (to the best of our knowledge).

[4]Bin-packing heuristics, including those used to assign tasks to processors, usually perform better with smaller items. As the mean task utilization increases, the performance of these heuristics tends to degrade, resulting in more schedulability loss. Although heavy tasks are rare, some techniques, such as hierarchical scheduling, can introduce heavy "server" tasks. Hence, it is unrealistic to assume that only light tasks will occur in practice.

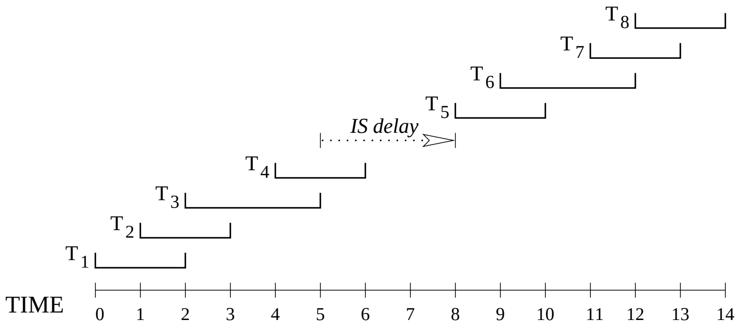


Figure 4: The PF-windows of the first eight subtasks of an IS task T with weight 8/11. Subtask T_5 is released three units late causing all later subtask releases to be delayed by three time units.

3 Intra-sporadic Task Systems

The *intra-sporadic* (IS) task model was proposed as an extension of the well-studied periodic and sporadic models [3, 20]. The sporadic model generalizes the periodic model by allowing *jobs* to be released "late", *i.e.*, the separation between consecutive job releases of a task is allowed to be more than the task's period. The IS model generalizes this by allowing *subtasks* to be released late, as illustrated in Figure 4. More specifically, the separation between subtask releases $r(T_i)$ and $r(T_{i+1})$ is allowed to be more than $\lfloor i/wt(T) \rfloor - \lfloor (i-1)/wt(T) \rfloor$, which would be the separation if T were periodic. Thus, an IS task is obtained by allowing a task's windows to be right-shifted from where they would appear if the task were periodic. Figure 4 illustrates this.

Under the IS model, each subtask T_i has an *offset*, denoted $\theta(T_i)$, that gives the amount by which its window has been right-shifted. Hence, (3) and (4) can be expressed as follows.

$$r(T_i) = \theta(T_i) + \left\lfloor \frac{i-1}{wt(T)} \right\rfloor \quad (7)$$

$$d(T_i) = \theta(T_i) + \left\lceil \frac{i}{wt(T)} \right\rceil \quad (8)$$

The offsets are constrained so that the separation between any pair of subtask releases is at least the separation between those releases if the task were periodic. Formally, the offsets must satisfy

$$k \geq i \Rightarrow \theta(T_k) \geq \theta(T_i). \tag{9}$$

Under the IS model, a subtask T_i is permitted to execute before the beginning of its Pfair window. That is, each subtask T_i has an eligibility time $e(T_i)$, as in ERfair scheduling, and $e(T_i)$ is allowed to be less than $r(T_i)$. The interval $[r(T_i), d(T_i))$ is said to be T_i's *PF-window*, while the interval $[e(T_i), d(T_i))$ is said to be its *IS-window*.

Using the definitions above, it is easy to show that sporadic and periodic tasks are special cases of IS tasks. In particular, using the expression $J(T_i) = \lfloor \frac{i-1}{T.e} \rfloor + 1$, which maps a subtask to the index (starting at 1) of the associated job, a periodic task T can be expressed as an IS task with $e(T_i) = T.\phi + (J(T_i) - 1) \cdot T.p$, *i.e.*, each grouping of $T.e$ subtasks become eligible simultaneously when the associated job is released. Similarly, all subtasks associated with a job under the sporadic model will become eligible when the associated job is released. All subtasks within a single job of a sporadic task have the same offset, *i.e.*, only the window of the first of these subtasks may be separated from that of its predecessor by an amount exceeding that in a periodic system.

The IS model allows the instantaneous rate of subtask releases to differ significantly from the average rate (given by a task's weight). Hence, it is more suitable than the periodic model for many applications, particularly those in networked systems. Examples include web servers that provide quality-of-service guarantees, packet scheduling in networks, and the scheduling of packet-processing activities in routers [25]. Due to network congestion and other factors, packets may arrive late or in bursts. The IS model treats these possibilities as first-class concepts and handles them more seamlessly. In particular, a late packet arrival corresponds to an IS delay. On the other hand, if a packet arrives early (as part of a bursty sequence), then its eligibility time will be less than its Pfair release time. Note that its Pfair release time determines its deadline. Thus, in effect, an early packet arrival is handled by postponing its deadline to where it would have been had the packet arrived on time.

Feasibility. In [3], Anderson and Srinivasan showed that an IS task set τ is feasible on M processors if and only if (5) holds. This feasibility proof is a straightforward extension of the feasibility proof for periodic tasks described earlier in Sec. 2.1.

Scheduling algorithms. In [20], Srinivasan and Anderson proved that PD^2 correctly schedules any feasible IS task system, and thus, is optimal for scheduling IS tasks. When prioritizing subtasks, the successor bits and group deadlines are calculated as if no IS delays occur in the future. For example, consider subtask T_5 in Figures 1 and 4. In Figure 1, T_5's group deadline is at time 8, while in Figure 4, its group deadline is at time 11. However, T_3's group deadline is at time 8 in both figures because T_3 is prioritized as if no IS delays occur in the future. In both figures, each of T_3 and T_5 has a b-bit of 1.

Theorem 4 ([20]) PD^2 *is optimal for scheduling intra-sporadic tasks on multiprocessors.*

Srinivasan and Anderson also showed that the earliest-pseudo-deadline-first (EPDF) algorithm, which uses only Rule (i) to prioritize subtasks, is optimal for scheduling IS tasks on M (> 1) processors if the weight of each task is at most $\frac{1}{M-1}$. (This result is fairly tight. In particular, if the weight of a task is allowed to be at least $\frac{1}{M-1} + \frac{1}{(M-1)^2}$, then EPDF can miss deadlines.)

Although the periodic model represents the worst-case behavior of an IS task, it turns out that the optimality proofs mentioned previously do not extend directly to the IS case. The primary reason for this is that the proofs are based on swapping arguments, which require *a priori* knowledge of the positions of future windows. To prove the optimality of PD^2, Srinivasan and Anderson developed a novel and effective lag-based argument that is more flexible than swapping-based arguments. (Interested readers are referred to [20] for the complete proof.) This same proof technique was also used to obtain the EPDF results mentioned above, and also Theorem 7 in the next section [22].

4 Dynamic Task Systems

In many real-time systems, the set of runnable tasks may change dynamically. For example, in an embedded system, different modes of operation may need to be supported; a mode change may require adding new tasks and removing existing tasks. Another example is a desktop system that supports real-time applications such as multimedia and collaborative-support systems, which may be initiated at arbitrary times. When considering dynamic task systems, a key issue is that of determining when tasks may join and leave the system without compromising schedulability.

4.1 Join and Leave Conditions

For IS task systems, a join condition follows directly from (5), *i.e.*, a task can join if the total weight of all tasks will be at most M after its admission. It remains to determine when a task may leave the system safely. (Here, we are referring to the time at which the task's share of the system can be reclaimed. The task may actually be allowed to leave the system earlier.) As shown in [8, 26], if a task with negative lag is allowed to leave, then it can re-join immediately and effectively execute at a rate higher than its specified rate, which may cause other tasks to miss their deadlines. Based on this observation, the conditions shown below are at least necessary, if not sufficient.

(C1) *Join condition*: A task T can join at time t if and only if (5) continues to hold after joining.[5]

Leave condition: A task T can leave at time t if and only if $t \geq d(T_i)$, where T_i is the last-scheduled subtask of T.

The condition $t \geq d(T_i)$ is equivalent to $lag(T, t) \geq 0$. To see why, note that since $t \geq d(T_i)$, task T receives at least i units of processor time in the ideal schedule by time t. Because T_i is the last-scheduled subtask of T in the actual schedule, T receives at most i quanta by time t. Hence, $lag(T, t) \geq 0$. As a straightforward extension of the feasibility proof for IS task systems [3], it can be easily shown that a feasible schedule exists for a set of dynamic tasks if and only if condition (C1) is satisfied. Indeed, condition (C1) has been shown to ensure schedulability when using *proportional-share* scheduling on a uniprocessor [8, 26]. However, as shown below, (C1) is not sufficient when using a priority-based Pfair algorithm (such as PF, PD, or PD^2) on a multiprocessor.

The theorem below applies to any "weight-consistent" Pfair scheduling algorithm. An algorithm is *weight-consistent* if, given two tasks T and U of equal weight with eligible subtasks T_i and U_j, respectively, where $i = j$ and $r(T_i) = r(U_j)$ (and hence, $d(T_i) = d(U_j)$), T_i has priority over a third subtask V_k if and only if U_j does. All known Pfair scheduling algorithms are weight-consistent.

Theorem 5 ([22]) *No weight-consistent Pfair scheduler can guarantee all deadlines on multiprocessors under* (C1)*.*

Proof: Consider task systems consisting of only two weight classes: class X with weight $w_1 = 2/5$ and class Y with weight $w_2 = 3/8$. Let $X_f = \{ T_1 \mid T \in X \}$ and $Y_f = \{ T_1 \mid T \in Y \}$. We construct a counterexample based upon which task weight is favored by the scheduler. (In each of our counterexamples, no subtask is eligible before its PF-window.) We say that X_f *is favored*

[5] If T joins at time t, then $\theta(T_1) = t$. A task that re-joins after having left is viewed as a new task.

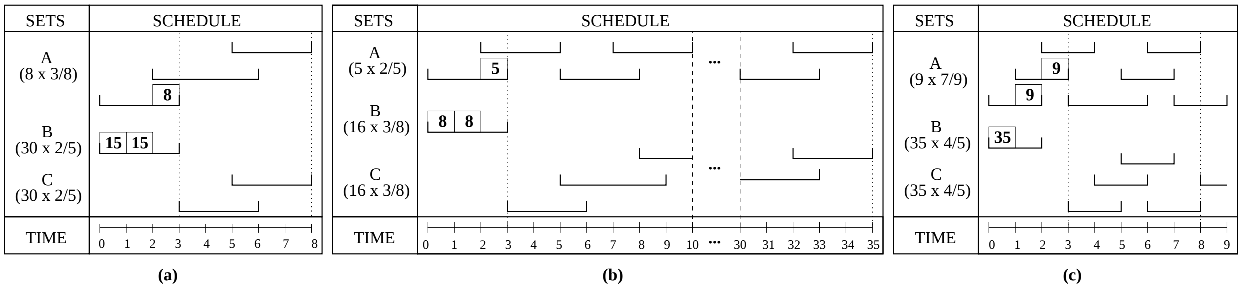


Figure 5: Counterexamples demonstrating insufficiency of (C1) and tightness of (C2). A boxed integer value n in slot t means that n of the subtasks from the corresponding task set are scheduled in that slot. The dotted vertical lines depict intervals with excess demand. **(a)** Theorem 5. Case 1: Tasks of weight 2/5 are favored at times 0 and 1. **(b)** Theorem 5. Case 2: Tasks of weight 3/8 are favored at times 0 and 1. **(c)** Theorem 6. The tasks of weight 4/5 are allowed to leave at time 3 and re-join immediately.

(analogously for Y_f) if, whenever subtasks in X_f and Y_f are released at the same time, those in X_f are given priority over those in Y_f.

Case 1: X_f is favored. Consider a 15-processor system containing the following sets of tasks.

> **Set A:** 8 tasks of weight w_2 that join at time 0.
>
> **Set B:** 30 tasks of weight w_1 that join at time 0 and leave at time 3.
>
> **Set C:** 30 tasks of weight w_1 that join at time 3.

Because $30w_1 + 8w_2 = 15$, this task system is feasible and the join condition in (C1) is satisfied. Furthermore, since $d(T_1) = \left\lceil \frac{5}{2} \right\rceil = 3$ for every $T \in B$, the leave condition in (C1) is also satisfied.

Since subtasks in X_f are favored, tasks in Set B are favored over those in Set A at times 0 and 1. Hence, the schedule for $[0, 3)$ will be as shown in Figure 5(a). Consider the interval $[3, 8)$. Each task in Sets A and C has two subtasks remaining for execution, which implies that the Set A requires 16 quanta and Set C requires 60 quanta by time 8. However, the total number of quanta in $[3, 8)$ is $15 \cdot (8 - 3) = 75$. Thus, one subtask will miss its deadline at or before time 8.

Case 2: Y_f is favored. Consider an 8-processor system containing the following sets of tasks.

Set A: 5 tasks of weight w_1 that join at time 0.

Set B: 16 tasks of weight w_2 that join at time 0 and leave at time 3.

Set C: 16 tasks of weight w_2 that join at time 3.

Because $5w_1 + 16w_2 = 8$, this task system is feasible and the join condition in (C1) is satisfied. Furthermore, since $d(T_1) = \left\lceil \frac{8}{3} \right\rceil = 3$ for every $T \in B$, the leave condition in (C1) is also satisfied.

Since subtasks in Y_f are favored, tasks in Set B are favored over those in Set A at times 0 and 1. Hence, the schedule for $[0, 3)$ will be as shown in Figure 5(b). Consider the interval $[3, 35)$. In this interval, each task in Set A requires $35 \cdot 2/5 - 1 = 13$ quanta. Similarly, each task in Set C requires $(35 - 3) \cdot 3/8 = 12$ quanta. However, the total requirement is $5 \cdot 13 + 16 \cdot 12 = 257$, whereas $[3, 35)$ contains only $(35 - 3) \cdot 8 = 256$ quanta. Thus, a deadline miss will occur at or before time 35. $\square$

The problem illustrated by the preceding proof is that subtasks are prioritized according to the requirements of their successors. When a subtask is the last subtask that a task will release, then it has no successors and hence should be given a successor bit of 0 and a trivial group deadline (*i.e.*, 0). However, without *a priori* knowledge of task departures, the scheduler must assign parameter values according to the normal priority rules, which are designed for persistent tasks. Indeed, Theorem 5 can be circumvented if the scheduler has such knowledge. For example, in Figure 5(a), if the scheduler had known that the subtasks in Set B had no successors, then it would have given those subtasks lower priority than those in Set A (by setting their successor bits to 0). However, in general, such *a priori* knowledge of task departures may not be available to the scheduler.

The examples in Figure 5(a)–(b) show that allowing a light task T to leave at $d(T_i)$ when $b(T_i) = 1$ can lead to deadline misses. We now consider heavy tasks.

Theorem 6 ([22]) *If a heavy task T is allowed to leave before $D(T_i)$, where T_i is the last-released subtask of T, then there exist task systems that miss a deadline under* PD^2.

Proof: Consider a 35-processor system containing the following sets of tasks (where $2 \leq t \leq 4$).

Set A: 9 tasks of weight 7/9 that join at time 0.

Set B: 35 tasks of weight 4/5 that join at time 0, release a subtask, and leave at time t.

Set C: 35 tasks of weight 4/5 that join at time t.

All tasks in Sets A and B have the same PD^2 priority at time 0, because each has a deadline at time 2, a successor bit of 1, and a group deadline at time 5. Hence, the tasks in Set B may be given higher priority.[6] Assuming this, Figure 5(c) depicts the schedule for the case of $t = 3$.

Consider the interval $[t, t + 5)$. Each task in Sets A and C has four subtasks with deadlines in $[t, t + 5)$ (see Figure 5(c)). Thus, $9 \cdot 4 + 35 \cdot 4 = 35 \cdot 5 + 1$ subtasks must be executed in $[t, t + 5)$. Since $35 \cdot 5$ quanta are available in $[t, t + 5)$, one subtask will miss its deadline. $\square$

The cases considered in the proofs of Theorems 5 and 6 motivate the new condition (C2), shown below. In [22], this condition is shown to be sufficient for ensuring the schedulability of dynamic IS task sets scheduled by PD^2. By Theorems 5 and 6, this condition is tight.

(C2) *Join condition*: A task T can join at time t if and only if (5) continues to hold after joining.

Leave condition: A task T can leave at time t if and only if $t \geq \max(D(T_i), d(T_i) + b(T_i))$, where T_i is the last-scheduled subtask of T.

Theorem 7 ([22]) PD^2 *correctly schedules any dynamic IS task set that satisfies* (C2).

Observe that (C2) guarantees that periodic and sporadic tasks can always leave the system at period boundaries. This follows from the fact that the last subtask T_i in each job will have a successor bit of 0, which implies that $\max(D(T_i), d(T_i) + b(T_i)) = \max(d(T_i), d(T_i) + 0) = d(T_i)$.

[6]This counterexample also works for PF and PD since both will give higher priority to the tasks in Set B at time 0.

4.2 QRfair Scheduling

In dynamic task systems, spare processing capacity may become available that can be consumed by the tasks present in the system. One way for a task T to consume such spare capacity is by early releasing its subtasks. However, by doing this, it "uses up" its future subtask deadlines. As a result, if the system load later increases, then T will be competing with other tasks using deadlines far into the future. Effectively, T is penalized when the load changes for having used spare capacity in the past.

In recent work, Anderson, Block, and Srinivasan proposed an alternative form of early-release scheduling called *quick-release* (*QRfair*) scheduling [4]. QRfair scheduling algorithms avoid penalizing tasks for using spare capacity by allowing them to shift their windows forward in time when an idle processor is detected. The benefits of doing this strongly resemble those provided by uniprocessor fair scheduling schemes based on the concept of *virtual time* [26].

5 Supertasking

In [16], Moir and Ramamurthy observed that the migration assumptions underlying Pfair scheduling may be problematic. Specifically, tasks that communicate with external devices may need to execute on specific processors and hence cannot be migrated. They further noted that statically binding tasks to processors may significantly reduce migration overhead in Pfair-scheduled systems.

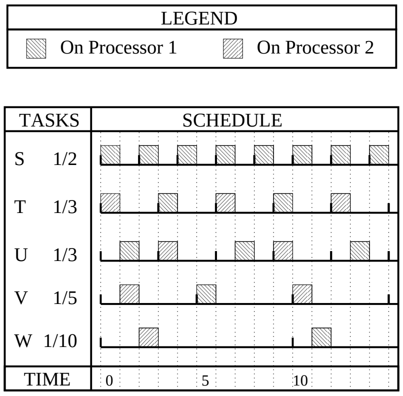
To support non-migratory tasks, they proposed the use of *supertasks*. In their approach, a supertask $\mathcal{S}_p$ replaces the set of tasks that are bound to processor p, which become the *component tasks* of $\mathcal{S}_p$. (We use $\mathcal{S}_p$ to denote both the supertask and its set of component tasks.) Each supertask $\mathcal{S}_p$ then competes with a weight equal to the cumulative weight of its component tasks, as shown below.

$$wt(\mathcal{S}_p) = \sum_{T \in \mathcal{S}_p} wt(T)$$

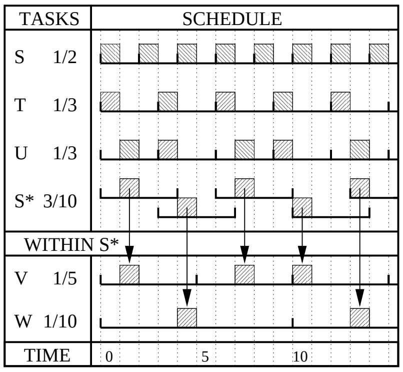
Whenever a supertask is scheduled, one of its component tasks is selected to execute according to an internal scheduling algorithm. Although Moir and Ramamurthy proved that EPDF is optimal for scheduling component tasks, they also demonstrated that component-task deadline misses may occur when using each of PF, PD, and PD^2 as the global scheduling algorithm.

Figures 6(a)–(b) illustrate supertasking. Figure 6(a) shows a PD^2 schedule in which two processors are shared among five tasks, labeled $S, \ldots, W$, that are assigned weights 1/2, 1/3, 1/3, 1/5, and 1/10, respectively. Vertical dashed lines mark the slot boundaries and boxes show when each task is scheduled. Figure 6(b) is derived from Figure 6(a) by placing tasks V and W into a supertask S^* that competes with weight $wt(S^*) = 3/10 = wt(V) + wt(W)$ and is bound to processor 2. The arrows show S^* passing its allocation (upper schedule) to one of its component tasks (lower schedule) based upon an EPDF prioritization. Although no component-task deadlines are missed in this example, such a weight assignment is *not* sufficient, in general, to guarantee that all deadlines are met when using the PD^2 algorithm [16].

Supertasking as a hierarchical-scheduling mechanism. In [11], Holman and Anderson considered supertasking more generally as a mechanism for achieving hierarchical scheduling in Pfair-scheduled systems. Hierarchal scheduling is particularly interesting (and desirable) under Pfair scheduling because many common task behaviors (*e.g.*, blocking and self-suspension) can disrupt the fair allocation of processor time to tasks, making scheduling more difficult. One way to compensate for these behaviors is to schedule a set of problematic tasks as a single entity and then use a second-level (component-task) scheduler to allocate the group's processor time to the member (component) tasks. In this approach, fairness is somewhat relaxed in that the group is required



(a) PD^2 schedule



(b) Supertasking PD^2 schedule

Figure 6: Sample Pfair schedules for a task set consisting of five tasks with weights 1/2, 1/3, 1/3, 1/5, and 1/10, respectively. **(a)** Normal schedule produced when no supertasks are used. **(b)** Schedule produced when tasks V and W are combined into the supertask S*, which competes with weight 3/10 and is bound to processor 2.

to make progress at a steady rate rather than each individual task within the group. (Since fair scheduling was introduced primarily to improve schedulability, weakening the fairness guarantee in order to improve schedulability should be an acceptable trade-off in many cases.)

One immediate advantage of this relaxed fairness is that the component-task scheduler need not be a fair scheduler. Indeed, using an unfair scheduler can result in substantial improvements, as discussed in [11] and as demonstrated in [12, 13]. In addition, using supertasking to achieve hierarchical scheduling ensures that only one component task is executing at each instant, *i.e.*, scheduling within a supertask is a single-resource problem. This characteristic provides two advantages. First, selectively preventing tasks that share resources from executing in parallel can reduce contention, and hence, improve the schedulability of the system, as demonstrated in [12, 13]. (This advantage is discussed in more detail in the next section.) Second, component-task schedulability can be analyzed using demand-based arguments and other simple uniprocessor techniques. Informally, a demand-based argument states that a deadline miss must be preceded by an interval of time during which the total time required to service all pending jobs (in priority order) exceeded the available processor time. Such analysis is desirable due both to its simplicity and also to the ease with which it can be understood and adapted to new situations. Unfortunately, demand-based arguments tend to be ineffective on multiprocessors because they do not account for parallel execution.

In the rest of this section, we provide an overview of problems and results relating to supertasking. We begin by discussing the scheduling problems that must be addressed. We conclude by briefly describing an open problem relating to supertasking.

Scheduling within a supertask. Since a supertask is effectively a single resource, an inductive swapping argument can be used to prove that both EPDF and EDF scheduling algorithms can schedule any periodic component-task set that is feasible under a given supertask allocation, provided that all component tasks are independent. We briefly sketch the proof for the EDF case below.

First, suppose, to the contrary of the claim, that a schedule S exists in which all job deadlines are met, but that the schedule produced by EDF, S^{EDF}, does not meet all deadlines. Also, let t be the earliest time (slot) at which the scheduling decisions differ. Specifically, select x and y so that $x \in S_t$, $x \notin S_t^{\text{EDF}}$, $y \notin S_t$, and $y \in S_t^{\text{EDF}}$. By the EDF prioritization, y must have a deadline that is at most the deadline of x. Furthermore, since $y \notin S_t$ and both schedules are identical up to time t, it follows that y must be scheduled next at some time t' in S, where $t' > t$. By the initial assumptions, no deadlines are missed in schedule S, which implies that y's (and hence x's) deadline must be at or after $t' + 1$. (Recall that job deadlines come at the end of slots while scheduling decisions are made at the start of each slot.) It follows that x and y can be swapped in schedule S without introducing a deadline miss. This process can then be repeated until all scheduling decisions at time t are identical in both schedules. By then inducting over time, all decisions in schedule S can be made to agree with those in S^{EDF} without introducing a deadline miss. However, this contradicts the claim that S^{EDF} contains a deadline miss, which completes the proof. Hence, EDF is an optimal policy when scheduling within a supertask. The proof is identical for EPDF, with the only exception being that subtask deadlines are considered instead of job deadlines.

Scheduling the supertask. The main problem when using supertasks is to schedule a supertask so that its component-task set is feasible. There are two basic approaches to solving this problem. First, the global scheduler could be modified to handle supertasks as a special class. When invoked, the global scheduler would first need to determine when each supertask should be scheduled to guarantee the schedulability of its component tasks and then schedule the remaining tasks "around" the supertasks. Unfortunately, it seems unlikely that supertasks could be granted such special treatment while maintaining the optimality of the global scheduler. In addition, this approach is unappealing in that it would likely increase scheduling overhead and would not provide a clean separation between global and component-task scheduling.

An alternative approach, which Holman and Anderson investigated in [11, 14], is to assign to each supertask a weight that is sufficient to guarantee component-task feasibility under *any* Pfair schedule. We refer to this weight-selection problem as the *reweighting* problem. Unfortunately, Moir and Ramamurthy's work implies that this approach will necessarily result in some schedulability loss, at least in some cases. In [11], Holman and Anderson presented rules for selecting a supertask weight. These rules are based upon demand-based analysis and stem from the theorem shown below, which bounds the amount of processor time available to a supertask over any interval of a given length.

Theorem 8 (Holman and Anderson) *A Pfair-scheduled supertask with weight w will receive at least $\lfloor wL \rfloor - 1$ quanta of processor time over any interval spanning L slots.*

The theorem stated below follows directly from the rules presented in [11] and highlights some of the interesting practical ramifications of supertasking.

Theorem 9 *All component tasks of a Pfair-scheduled supertask $\mathcal{S}$ will meet their deadlines if*

$$wt(\mathcal{S}) = \min\left(1, \sum_{T \in \mathcal{S}} wt(T) + \frac{1}{L}\right),$$

provided that either **(i)** *component tasks are scheduled using EPDF and L is the* shortest window length *of any component task, or* **(ii)** *component tasks are scheduled using EDF and L is the* smallest period *of any component task.*

The first implication of the above theorem is that EDF is less costly than EPDF when scheduling component tasks. Although this would seem to contradict the earlier claim that both algorithms are optimal in this context, it actually highlights the fundamental difference between fair and unfair scheduling. Specifically, EDF is optimal for scheduling when *job* deadlines must be met (*i.e.*, for periodic-task scheduling), but is not optimal when *subtask* deadlines must be met (*i.e.*, for Pfair scheduling). Since the Pfairness constraint is stricter than the periodic-task constraint, it is more costly to provide this guarantee when scheduling component tasks. Hence, EDF should be used for scheduling component tasks whenever fairness is not essential to the correctness of the system.

The second implication is that the schedulability loss that stems from reweighting a supertask is bounded (by $\frac{1}{L}$) and should often be small in practice. For instance, if the quantum size is 5 milliseconds and the smallest component-task period is 250 milliseconds, the inflation will be at most 0.02. In addition, this penalty can be reduced when using EDF by increasing component-task periods.

Feasibility. Although sufficient schedulability conditions have been derived for hierarchical scheduling with supertasks (in [11, 14]), the corresponding feasibility problem (that is, the problem of devising necessary and sufficient schedulability conditions) remains open. In [16], Moir and Ramamurthy proved that a periodic task system containing non-migratory tasks is feasible if and only if it satisfies

$$\sum_{T \in \tau} wt(T) \leq M \wedge \left(\forall p : 1 \leq p \leq M : \sum_{T \in \mathcal{S}_p} wt(T) \leq 1 \right).$$

This proof is based on a flow-graph construction similar to that used by Baruah *et al.* [6] to prove the feasibility condition (5) for periodic task systems. Unfortunately, this proof does not appear to extend to the supertasking approach. Consequently, there is no known feasibility condition for systems in which periodic tasks are scheduled within Pfair-scheduled supertasks.

6 Resource Sharing

In this section, we discuss techniques for supporting task synchronization under Pfair scheduling, including techniques for lock-free [12] and lock-based [13] synchronization.

6.1 Lock-free Synchronization

In [12], Holman and Anderson considered Pfair-scheduled systems in which lock-free shared objects are used. Lock-free algorithms synchronize access to shared software objects while ensuring system-wide liveness, even when faced with process halting failures. Specifically, lock-free algorithms guarantee that *some* task is always capable of making progress. Lock-based algorithms cannot satisfy this property since a task can prevent all other tasks from making progress by halting while holding a lock.

Lock-free algorithms typically avoid locking through the use of "retry loops" and strong synchronization primitives. Figure 7 depicts a lock-free enqueue operation. An item is enqueued in this

typedef *Qtype*:
    record *data*: *valtype*; *next*: **pointer to** *Qtype*

shared var
    Head, *Tail*: **pointer to** *Qtype*

private var
    old, *new*: **pointer to** *Qtype*; *input*: *valtype*;
    addr: **pointer to pointer to** *Qtype*

procedure `Enqueue`(*input*)
    $*new := (input, nil)$;
    do $old := Tail$;
        if $old \neq nil$ **then** $addr := \&((*old).next)$
        else $addr := \&Head$ **fi**
    while $\neg$`CAS2`$(\&Tail, addr, old, nil, new, new)$

Figure 7: This figure shows a lock-free enqueue operation using the `CAS2` primitive. The first two parameters of `CAS2` specify addresses of two shared variables, the next two parameters are values to which these variables are compared, and the last two parameters are new values to assign to the variables if both comparisons succeed. Although `CAS2` is uncommon, it makes for a simple example here.

implementation by using a *two-word compare-and-swap* (`CAS2`) instruction to atomically update a tail pointer and either the "next" pointer of the last item in the queue or a head pointer, depending on whether the queue is empty. This loop is executed repeatedly until the `CAS2` instruction succeeds. An important property of lock-free implementations such as this is that operations may *interfere* with each other. In this example, an interference results when a successful `CAS2` by one task causes another task's `CAS2` to fail.

Blocking due to lock requests complicates Pfair scheduling considerably, as explained below. Because of this, the lock-free approach will likely perform better when it is applicable.

Problems with lock-based synchronization. Locking is problematic in Pfair-scheduled systems for three reasons. First, Pfair scheduling can lead to long delays. Specifically, low-weight tasks are scheduled infrequently, as shown in Figure 8. In general, the worst-case delay experienced by lock-requesting tasks due to a lock-holding Pfair-scheduled task T is inversely-proportional to $T.w$. Second, critical sections can span multiple subtasks. Hence, a task's priority may change *during* one of its critical sections. Most existing locking protocols implicitly assume that each task's priority is static, at least until its critical section completes. Third, Pfair weights are a form of *reservation* mechanism, which complicates blocking-time accounting.

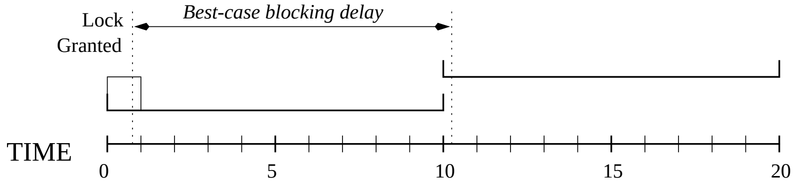
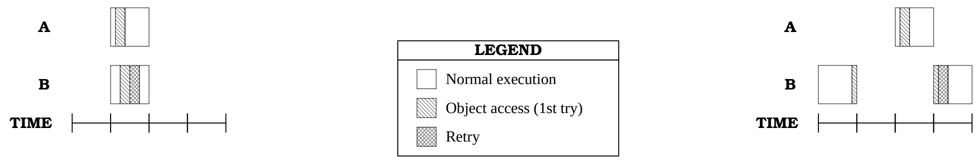


Figure 8: This figure illustrates the blocking delay bounds implied by Pfair scheduling for a task T of weight 1/10. In this example, task T obtains a lock within slot 0 and is preempted before releasing it. Assuming the lock will be released shortly after the start of the next quantum allocated to T, the shared resource will still be unavailable for an interval of at least 9 slots.

Accounting for lock-free overhead. Lock-free algorithms are usually viewed as impractical for real-time multiprocessor systems because bounding the worst-case number of retries is difficult. However, the tight synchrony provided by Pfair scheduling facilitates worst-case analysis. Specifically, Pfair scheduling guarantees that quantum boundaries align across processors and that at most M tasks execute within each time slot. Hence, a scheduled task faces interference from no more than $M - 1$ other tasks in each time slot in which it executes. In addition, Pfair's tight synchrony implies that interference takes one of two possible forms: interference within a quantum and interference across multiple quanta. Both forms are illustrated in Figure 9.

Bounding the number of interferences across multiple quanta is trivial in most cases due to practical considerations. In experiments conducted by Ramamurthy [19] on a 66 MHz processor, operation 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, it is unlikely that any shared-object operation will be preempted more than once. Hence, we only consider using lock-free techniques when all operations are guaranteed to be preempted at most once, which is expected often to be the case. Therefore, no more that one retry is needed due to interference across multiple quanta.



(a) Interference within a quantum **(b) Interference across quanta**

Figure 9: This figure shows scenarios under which task A causes task B's lock-free operation to fail, and hence be retried. **(a)** A is scheduled in parallel with B. **(b)** B is preempted during the operation.

Now, consider a specific task T that performs an operation on a lock-free object ρ. Let $Q(T, \rho)$ denote the worst-case number of operations applied to ρ by any group of $M - 1$ tasks, excluding T, within one quantum, *i.e.*, $Q(T, \rho)$ is an upper bound on the number of interferences experienced by T within any single quantum. Since T's operation is preempted at most once, it follows that T's operation is interfered with at most $2 \cdot Q(T, \rho) + 1$ times before it operation completes. Hence, T's operation will require no more than $2 \cdot Q(T, \rho) + 1$ retries.

Reducing retry overhead. Recall that interference within a quantum can only occur when two or more tasks that share a common object are scheduled in parallel. One simple technique for reducing such interferences is to place tasks that share common objects into the same supertask, thereby reducing the value of $Q(T, \rho)$. For instance, consider a 4-processor system in which four tasks, denoted $A, \ldots, D$, share an object. Further, suppose that $A, \ldots, D$ make at most 1, 4, 3, and 6 accesses to the object within a single quantum. Without supertasking, task A can experience up to $6+4+3 = 13$ interferences within each quantum due to tasks $B, \ldots, D$. The worst-case scenario occurs when $A, \ldots, D$ are all scheduled in the same time slot. However, if $B, \ldots, D$ are placed within a supertask, task A can experience no more than $\mathsf{max}(6, 4, 3) = 6$ interferences within each quantum since A can be scheduled together with at most one task from the supertask.

Uniprocessor object implementations. Another potential benefit of supertasking is the ability to utilize *uniprocessor* lock-free algorithms. Such algorithms can be used in the special case in which all tasks that access an object are contained in the same supertask. Uniprocessor implementations tend to be structurally simpler than their multiprocessor counterparts and hence more efficient. In addition, strong synchronization primitives can often be efficiently implemented *in software* on a uniprocessor [1], which avoids the need for hardware support.

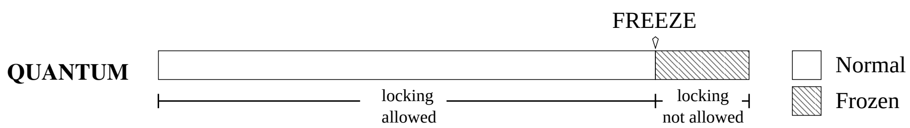
Evaluating the supertasking trade-off. Unfortunately, supertasking's benefits may be counterbalanced by the schedulability loss due to reweighting. To determine whether supertasking is a viable tool for reducing lock-free overhead, Holman and Anderson conducted a series of experiments for 2-, 4-, 8-, and 16-processor systems [12]. In each experiment, a task set that shares lock-free objects was randomly generated and then the cumulative weight of the system was computed both with and without supertasks. (The heuristic used to assign tasks to supertasks and the experimental setup are described in detail in [12].) For 2-processor systems, schedulability loss due to reweighting was found to outweigh the reduction in lock-free overhead in the vast majority (91%) of the cases considered. This is because interference within a quantum is already limited to only a single interfering task per time slot. Hence, the use of supertasking provides only marginal benefits. On the other hand, supertasking improved schedulability in 82.5%, 98.8%, and 99.9% of the task sets considered for the 4-, 8-, and 16-processor systems, respectively.

6.2 Locking Synchronization

In [13], Holman and Anderson presented two approaches for supporting (non-nested) critical sections under Pfair scheduling. In this section, we summarize the concepts underlying these approaches. The analysis of systems using these approaches is straightforward; details can be found in [13].

Limitations of lock-free synchronization. Although lock-free algorithms avoid many problems that come with locking, they have many limitations. As previously mentioned, lock-free algorithms often require strong synchronization primitives. Such primitives may not be available in some multiprocessor systems. In addition, time and space overheads do not scale well with respect to object complexity. Complex objects can usually be implemented much more efficiently using locking. Finally, lock-free techniques can only be applied to objects implemented in software. Locks are still needed to synchronize access to external devices.

Approach 1: Timer-based. In the first approach presented by Holman and Anderson, the durations of all critical sections guarded by the lock in question are assumed to be much shorter than the length of a scheduling quantum. (For reasons already explained, this approach is expected to be widely applicable.) For cases in which this assumption holds, a per-lock timer signal, called FREEZE (see the diagram below), can be used that occurs towards the end of each time slot. Once this signal is received, lock granting is disabled. By placing this signal appropriately in time, a task that risks being preempted while holding the lock can be prevented from obtaining it. Hence, no locks using this implementation can be held across a quantum boundary, which avoids many of the previously-mentioned problems associated with locking. Effectively, the blocking overhead caused by the preemption of a lock-holding task is traded for blocking overhead caused by the frozen intervals. We discuss the effects of this trade-off at the end of this section.



Approach 2: Server-based. This approach can be used to implement any lock, *i.e.*, no assumptions are made about critical-section durations. In this approach, a server executes all crit-

ical sections guarded by a specific lock in place of the requesting task. The primary alternative to a server-based approach is the use of some form of inheritance mechanism. Unfortunately, inheritance-based protocols are problematic under Pfair scheduling as explained below.

Under an inheritance-based protocol, a lock-holding task is allowed to *inherit* some of the scheduling parameters of a lock-requesting task that it blocks. For example, under the priority inheritance protocol [18], a lock-holding task T inherits the *priority* of a higher-priority task that it blocks. Under Pfair scheduling, a task T that inherits another task U's scheduling parameters is temporarily bound to U's state. Thus, an explicit distinction is made between scheduling states and the tasks that are bound to them. Usually a task T is bound to its own state, but when inheritance occurs, this binding may change. Specifically, T may become bound to another task U's state, or even to *both* its own state *and* U's state, leaving U temporarily bound to no state.

The problem with using inheritance-based protocols in Pfair-scheduled systems is that blocking durations are both determined by task weights *and* compensated for by changing task weights. Specifically, the blocking experienced by a task T, which is determined by the weights of tasks that share resources with T, is compensated for by adjusting T's weight. However, changing T's weight changes its Pfair window layout, which may either increase or decrease the worst-case delay before a lock held by T is released. This change in delay may then require compensatory changes in other tasks' weights that share locks with T. Unfortunately, when these other weights change, the worst-case blocking experienced by T will change also, requiring T to change weight again. Identifying an optimal weight assignment is a non-trivial task under inheritance-based protocols. Furthermore, changing the weights of tasks may also change the worst-case blocking scenario, which further complicates the weight assignment process. Ultimately, the interdependence of weights caused by inheritance-based protocols makes the effect of a weight change difficult to predict, which may be undesirable even if the weight-selection algorithm's overhead is reasonable.

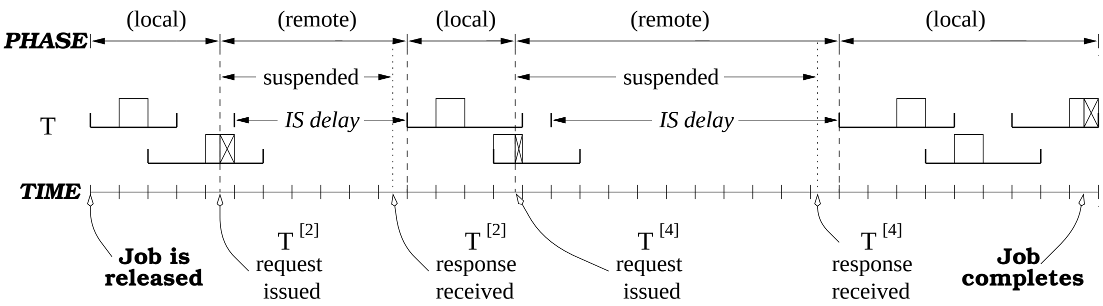


Figure 10: Behavior of a lock-requesting task T with weight 7/19 under the SWSP. The figure shows the release pattern for T's windows, where T has five phases per job. The first, third, and fifth phases consist of local work, while the second and fourth phases require locks and are hence executed remotely by the servers. Phase transitions are shown across the top of the figure and time is shown across the bottom. Arrows show when each RPC begins and ends. Unshaded boxes show where T executes locally while boxes containing X's denote unutilized processor time.

The protocol proposed by Holman and Anderson, referred to as the Static-Weight Server Protocol (SWSP), uses statically-weighted lock servers to execute critical sections in place of the requesting task [13]. Effectively, critical sections guarded by a common lock are implemented as remote procedure calls serviced by a common server. Delays caused by critical-section requests are easily modelled and accounted for by using the IS task model, as illustrated in Figure 10. In addition, the use of statically-defined server weights makes blocking-term computations straightforward and the responsiveness of the server more predictable.

Comparison. Holman and Anderson conducted breakdown utilization tests for systems with 2, 4, 8, 16, and 32 processors to evaluate the schedulability loss under each of these protocols [13]. As expected, they found that the timer-based approach performs and scales very well when critical sections are much shorter than the quantum's duration. However, breakdown utilizations dropped off quickly once critical section durations longer than approximately one-quarter of the quantum's duration were permitted. The SWSP, on the other hand, had typical breakdown utilizations ranging from 75% to 25% of the system's total utilization. However, in the context of long critical sections, the SWSP *did* outperform the timer-based approach in many cases.

7 Summary

In this chapter, we discussed the basic concepts behind Pfair scheduling and the various extensions that have been proposed in the literature. We described the three optimal Pfair algorithms: PF, PD, and PD^2. The PD^2 algorithm is optimal for scheduling tasks under the IS model, which generalizes both the periodic and sporadic models. We also discussed extensions that support dynamic task sets, hierarchical scheduling, and resource sharing.

References

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

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

[3] J. Anderson and A. Srinivasan. Pfair scheduling: Beyond periodic task systems. In *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications*, pp. 297–306, December 2000.

[4] J. Anderson, A. Block, and A. Srinivasan. Quick-release fair scheduling. In submission, May 2003.

[5] S. Baruah. Fairness in periodic real-time scheduling. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pp. 200–209, December 1995.

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

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

[8] S. Baruah, J. Gehrke, C.G. Plaxton, I. Stoica, H. Abdel-Wahab, and K. Jeffay. Fair on-line scheduling of a dynamic set of tasks on a single resource. *Information Processing Letters*, 26(1):43–51, January 1998.

[9] A. Chandra, M. Adler, and P. Shenoy. Deadline fair scheduling: Bridging the theory and practice of proportionate-fair scheduling in multiprocessor servers. In *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium*, pp. 3–14, May 2001.

[10] L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

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

[12] P. Holman and J. Anderson. Object sharing in Pfair-scheduled multiprocessor systems. In *Proceedings of the 13th Euromicro Conf. on Real-Time Systems*, pp. 111–122, June 2002.

[13] P. Holman and J. Anderson. Locking in Pfair-scheduled multiprocessor systems. In *Proceedings of the 23rd IEEE Real-time Systems Symposium*, pp. 149–158, December 2002.

[14] P. Holman and J. Anderson. Using hierarchal scheduling to improve resource utilization in multiprocessor real-time systems. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, to appear, July 2003.

[15] P. Holman and J. Anderson. The staggered model: Improving the practicality of Pfair scheduling. In submission, May 2003.

[16] M. Moir and S. Ramamurthy. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pp 294–303, December 1999.

[17] S. Ramamurthy and M. Moir. Static-priority periodic scheduling of multiprocessors. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, pages 69–78, December 2000.

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

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

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

[21] A. Srinivasan and J. Anderson. Efficient scheduling of soft real-time applications on multiprocessors. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, to appear, July 2003.

[22] A. Srinivasan and J. Anderson. Fair scheduling of dynamic task systems on multiprocessors. In *Proceedings of the 11th International Workshop on Parallel and Distributed Real-Time Systems*, April 2003.

[23] A. Srinivasan, P. Holman, and J. Anderson. Integrating aperiodic and recurrent tasks on fair-scheduled multiprocessors. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, pp. 19–28, June 2002.

[24] A. Srinivasan, P. Holman, J. Anderson, and S. Baruah. The case for fair multiprocessor scheduling. In *Proceedings of the 11th International Workshop on Parallel and Distributed Real-Time Systems*, April 2003.

[25] A. Srinivasan, P. Holman, J. Anderson, S. Baruah, and J. Kaur. Multiprocessor scheduling in processor-based router platforms: Issues and ideas. In *Proceedings of the 2nd Workshop on Network Processors*, pages 48–62, February 2002.

[26] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C.G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 288–299, December 1996.