Early-Release Fair Scheduling*

James H. Anderson and Anand Srinivasan

Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175

Phone: (919) 962-1757 Fax: (919) 962-1799 E-mail: {anderson,anands}@cs.unc.edu

December 1999

Abstract

We present a variant of Pfair scheduling, which we call *early-release fair* (ERfair) scheduling. Like conventional Pfair scheduling, ERfair scheduling algorithms can be applied to optimally schedule periodic tasks on a multiprocessor system in polynomial time. However, ERfair scheduling differs from Pfair scheduling in that it is *work conserving*. As a result, average job response times may be much lower under ERfair scheduling than under Pfair scheduling, particularly in lightly-loaded systems. In addition, runtime costs are lower under ERfair scheduling. This is because, in Pfair-scheduled systems, significant bookkeeping information is required to determine when a job of a task is and is not eligible for execution. In an ERfair system, this bookkeeping information is not required because, once released, a job continues to be eligible until it completes. To the best of our knowledge, ERfair scheduling is the first truly work-conserving scheduling discipline for periodic task systems that is optimal for multiprocessors.

Keyword: fairness, multiprocessors, optimality, Pfair, real-time scheduling, work-conserving schedulers.

*Work supported by NSF grants CCR 9732916 and CCR 9972211. The first author was also supported by an Alfred P. Sloan Research Fellowship.

1 Introduction

A major step forward in the evolution of processor scheduling techniques was recently achieved in the work of Baruah and colleagues on *Pfair scheduling* [3, 5]. Pfair scheduling differs from more conventional real-time scheduling disciplines in that tasks are explicitly required to make progress at steady rates. In most real-time scheduling schemes, the notion of a rate is implicit. For example, in the classic periodic task model, each task T executes at a rate given by $T.e/T.p$, where $T.e$ is the *execution cost* of each job (i.e., instance) of T, and $T.p$ is the *period* of T. However, this notion of a rate is a bit inexact: during each interval of length $T.p$, there are no guarantees as to exactly *which* $T.e$ time units will be allocated to task T. In particular, a job of T may be allocated $T.e$ time units at the beginning of its period, or at the end of its period, or its computation may be spread out more evenly. Under Pfair scheduling, this implicit notion of a rate is strengthened to require each task to be executed at a rate that is uniform across each job.

Executing tasks at steady rates has important consequences. For instance, the Pfair scheduling algorithms proposed by Baruah et al. *optimally* solve the problem of scheduling periodic tasks on a multiprocessor system in polynomial time. This is a problem that was previously viewed by most researchers as being almost undoubtedly *NP*-hard. Pfair scheduling algorithms schedule tasks by breaking them into quantum-length "subtasks" that are subject to intermediate deadlines. By breaking tasks into smaller executable units, Pfair scheduling algorithms circumvent many of the bin-packing-like problems that lie at the heart of intractability results involving multiple-resource real-time scheduling problems. Intuitively, it is easier to evenly distribute small, uniform items among the available bins than larger, nonuniform items.

Baruah et al. presented two Pfair scheduling algorithms called PF and PD [3, 5]. The two algorithms differ in the way in which ties are broken when two subtasks have the same deadline. In PF, ties are broken by comparing future subtask deadlines, which is somewhat expensive. In PD, ties are broken in constant time by inspecting four tie-break parameters. In recent work, we proved that PD can be simplified to use only two tie-break parameters [1]. (When we henceforth refer to PD, we mean the simplified algorithm described in [1].)

In this paper, we present a variant of Pfair scheduling, which we call *early-release fair* (ERfair) scheduling. ERfair scheduling differs from Pfair scheduling in a rather simple way. In Pfair scheduling, each subtask of a task T must execute within a "window" of time slots, the last of which is its deadline. If some subtask of T executes "early" within its window, then T is ineligible for execution until the beginning of the window of its next subtask. Thus, Pfair scheduling is not a true work-conserving discipline. A scheduler is *work conserving* if and only if it never leaves a processor idle when there exist uncompleted jobs in the system that could execute on that processor. Under ERfair scheduling, if two subtasks are part of the same job, then the second subtask becomes eligible for execution as soon as the first completes. In other words, a subtask may be released "early," i.e., before the beginning of its Pfair window. In an ERfair-scheduled system, no processor is ever idle while there exist uncompleted jobs to schedule. Thus, ERfair scheduling is work conserving.

In this paper, we show that an early-release version of PD, which we call ER-PD, can be used to optimally schedule tasks in an ERfair system. Because ER-PD is work conserving, it has several advantages over PD. In particular, average job response times may be much lower when using ER-PD than when using PD, especially in lightly-loaded systems. This makes ER-PD the preferred choice for applications in which average-case response requirements are stipulated in addition to worst-case response requirements. In addition, runtime costs are lower with ER-PD than with PD. This is because significant bookkeeping information must be maintained by PD to determine when a job of a task is and is not eligible for execution. With ER-PD, this bookkeeping information is not required. Finally, ER-PD supports a more flexible notion of a "rate" than PD. For example, if each subtask of a task is a routine that processes incoming packets, then with ER-PD, we can often avoid having to buffer for later processing packets that arrive "early" (e.g., due to jitter). As explained later, our notion of ERfair scheduling also can be applied within a task model that allows sporadic "separations" between subtask releases (and also job releases). In this model, ER-PD is a *multiprocessor* algorithm that functions in much the same way as the *uniprocessor* deadline scheduling algorithm proposed by Jeffay and Goodard for their rate-based execution (RBE) model [4].

In proving that ER-PD is correct, we borrow heavily from the correctness proof given by us for PD in [1]. This proof is quite long. Fortunately, much of the reasoning presented in [1] remains intact and can be encapsulated as a lemma. Given this lemma, only a modest number of additional cases must be considered. Interestingly, our correctness proof for ER-PD actually shows that a variety of "hybrid" optimal schedulers exist that incorporate aspects of both PD and ER-PD. For example, it is possible to declare certain tasks to be "early releasable" and others not. This might be useful if a small subset of tasks in a system are subject to stringent average response-time requirements. It is also possible to dynamically decide whether to early release subtasks or not, and to subject early releases to a threshold (e.g., a subtask may be allowed to release early, but only up to two time slots before its Pfair window). Such flexibility might be useful in systems in which average response-time requirements must be balanced against jitter requirements.

The rest of this paper is organized as follows. In Section 2, we define Pfair and ERfair scheduling. Then, in Section 3, we present the PD and ER-PD algorithms (the two algorithms differ only in when a subtask is considered to be eligible for execution). We prove that ER-PD is correct in Section 4. In Section 5, we present the results from some experiments we conducted to compare the runtime overhead associated with ER-PD with that of PD. We end with concluding remarks in Section 6.

2 Pfair and ERFair Scheduling

Consider a collection of periodic real-time tasks to be executed on a system of multiple processors. We assume that processor time in such a system is allocated in discrete time units, or quanta; the time interval $[t, t+1)$, where t is a nonnegative integer, is called *slot* t. Associated with each task T is a *period* $T.p$ and an *execution cost* $T.e$. Every $T.p$ time units, a new invocation of T with a cost of $T.e$ time units is released into the system; we call such an invocation a *job* of T. Each job of a task must complete execution before the next job of that task begins. Thus, $T.e$ time units must be allocated to T in each interval $[k \cdot T.p, (k+1) \cdot T.p)$, where $k \geq 0$. T

may be allocated time on different processors in such an interval, as long as it is not allocated time on different processors at the same time.

The sequence of allocation decisions over time defines a "schedule." Formally, a *schedule* S is a mapping $S : \tau \times N \mapsto \{0, 1\}$, where τ is a set of periodic tasks and N is the set of natural numbers. If $S(T, t) = 1$, then we say that *T is scheduled at slot t*. S_t denotes the set of tasks scheduled in slot t. The statements $T \in S_t$ and $S(T, t) = 1$ are equivalent.

Following Baruah et al. [3], we refer to $T.e/T.p$ as the *weight* of task T. We assume each task's weight is strictly less than one — a task with weight one would require a dedicated processor, and thus is quite easily schededuled. A task's weight defines the rate at which it is to be scheduled. Because processor time is allocated in quanta, we cannot guarantee that a task T will execute for *exactly* $(T.e/T.p)t$ time during each interval of length t. Instead, in a Pfair-scheduled system, processor time is allocated to each task T in a manner that ensures that its rate of execution never deviates too much from that given by its weight $T.e/T.p$. More precisely, correctness is defined by focusing on the *lag* between the amount of time allocated to each task and the amount of time that would be allocated to that task in an ideal system with a quantum approaching zero. Formally, the *lag of task T at time t*, denoted $lag(T, t)$, is defined as follows:

$$lag(T, t) = (T.e/T.p)t - allocated(T, t), \tag{1}$$

where $allocated(T, t)$ is the amount of processor time allocated to T in $[0, t)$. A schedule is *Pfair* if and only if

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

Informally, the allocation error associated with each task must always be less than one quantum.

Our notion of early-release scheduling is obtained by simply dropping the -1 lag constraint. Formally, a schedule is *early-release fair* (*ERfair*) if and only if

$$(\forall T, t :: lag(T, t) < 1). \tag{3}$$

Note that any Pfair schedule is ERfair, but not necessarily vice versa. It is straightforward to show that any ERfair schedule (and hence, any Pfair schedule) is periodic. In particular, in an ERfair schedule, $lag(T, t) = 0$ for $t = 0, T.p, 2T.p, 3T.p, \ldots$. This is because, for these values of t, $(T.e/T.p)t$ is an integer, and therefore by (1), $lag(T, t)$ is an integer as well. By (3), if $lag(T, t)$ is an integer, then it must be 0 or some negative integer. However, it cannot be a negative integer because this would imply that more processor time has been allocated to T than has been requested by jobs of T up to time t. Hence, $lag(T, t)$ is 0 for these values of t.

Baruah et al. [5] showed that a periodic task set τ has a Pfair schedule on M processors if and only if

$$\sum_{T \in \tau} \frac{T.e}{T.p} \leq M. \tag{4}$$

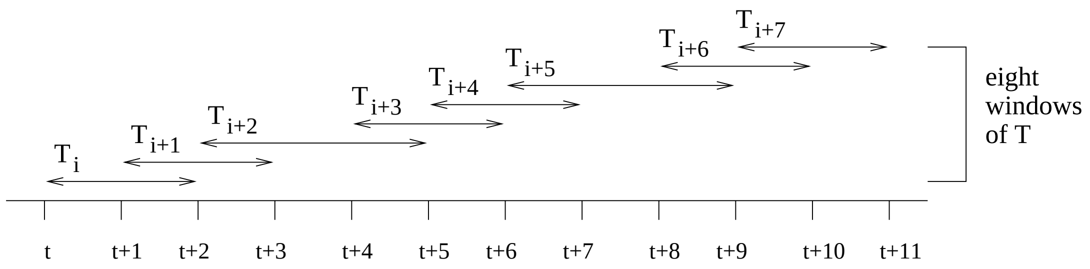

Figure 1: The eight "windows" of a task T with weight $T.e/T.p = 8/11$. Each of T's eight units of computation must be allocated processor time during its window, or else a lag-bound violation will result.

Because every Pfair schedule is also an ERfair schedule, (4) is a feasibility condition for ERfair systems as well.

The Pfair lag bounds given in (2) have the effect of breaking each task T into an infinite sequence of unit-time *subtasks*.[1] We denote the i^{th} subtask of task T as T_i, where $i \geq 1$. As in [3], we associate with each subtask T_i a *pseudo-release*

$$r(T_i) = \left\lfloor \frac{(i-1) \cdot T.p}{T.e} \right\rfloor \tag{5}$$

and a *pseudo-deadline*

$$d(T_i) = \left\lceil \frac{i \cdot T.p}{T.e} \right\rceil - 1. \tag{6}$$

(Derivations of these expressions can be found in [1].) In a Pfair-scheduled system, $r(T_i)$ is the first slot into which T_i could potentially be scheduled, and $d(T_i)$ is the last such slot. For brevity, we often refer to pseudo-deadlines and pseudo-releases as simply deadlines and releases, respectively. The interval $[r(T_i), d(T_i)]$ is called the *window* of subtask T_i and is denoted by $w(T_i)$. The *length* of window $w(T_i)$, denoted by $|w(T_i)|$, is defined as $d(T_i) - r(T_i) + 1$. A window spanning n time slots is called an *n-window*. As an example, consider a task T with weight $T.e/T.p = 8/11$. Each job of this task consists of eight windows, one for each of its unit-length subtasks. Using Equations (5) and (6), it is possible to show that the windows within each job of T are as depicted in Figure 1. Note that successive windows of T overlap by one slot and are of two different lengths. In general, consecutive windows of a task are either disjoint or overlap by one slot. In addition, either all windows of a task are of the same length, or they are of two different lengths (this property is proved in [1]).

Pfair and ERfair systems differ only in when a subtask is considered to be eligible for execution. In a Pfair system, a subtask T_i is eligible at time t if $t \in w(T_i)$ and if T_{i-1} has been scheduled prior to t but T_i has not. In an ERfair system, if T_i and T_{i+1} are part of the same job, then T_{i+1} becomes eligible for execution immediately after T_i executes; if T_i is the first subtask of its job, then it becomes eligible at the slot that begins its window (as in a Pfair system). The difference between Pfair- and ERfair-scheduled systems is illustrated in Figure 2. This figure shows Pfair and ERfair schedules for a two-processor system consisting of a set A of four tasks with weight 4/16 and set B of 16 tasks with weight 1/16. Under Pfair scheduling, each set-A job consists of four

[1]We have refrained from introducing the term "subjob" to refer to an invocation of a subtask because each subtask is invoked only once (each task consists of an *infinite* sequence of subtasks).

```
slot number:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15

                                            2  2  _  _
                                2  2  _  _
                    2  2  _  _
 A(4x4/16):   2  2  _  _

B(16x1/16):   _  _  2  2  _  _  2  2  _  _  2  2  _  _  2  2

                                (a)
```

```
slot number:  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15

                                2  2  _  _  _  _  _  _  _  _
                          2  2  _  _  _  _  _  _
                    2  2  _  _  _  _
 A(4x4/16):   2  2  _  _

B(16x1/16):   _  _  _  _  _  _  _  _  2  2  2  2  2  2  2  2

                                (b)
```

Figure 2: A schedule for a task set under Pfair scheduling **(a)** and under ERfair scheduling **(b)**. In this figure, tasks of a given weight are shown together. Each window is shown on a separate line and is depicted by showing the time slots it spans. Each column corresponds to a time slot. A slot t within a window is denoted by either an integer value or a dash. An integer value n means that n of the subtasks that must execute within that window are scheduled in slot t. A dash means that no such subtask is scheduled in slot t.

disjoint 4-windows, and each set-B job conists of one 16-window. Note that in the ERfair schedule (inset (b)), all set-A jobs have finished by time slot 7, while in the Pfair schedule (inset (a)), some of these jobs do not finish until time slot 13.

3 PD and ER-PD

In this section, we describe the PD scheduling algorithm for Pfair systems, and then the related algorithm ER-PD for ERfair systems. In PD, eligible subtasks are prioritized by their deadlines. If two subtasks have the same deadline, then two tie-break parameters are inspected to break the tie. (As explained earlier, the original PD algorithm [5] used four tie-break parameters. The version of PD considered here is the more efficient algorithm described in [1].) In the following paragraphs, we describe these tie-break parameters.

The first tie-break parameter is a bit that is associated with each subtask deadline. By Equations (5) and (6), $r(T_{i+1})$ is either $d(T_i)$ or $d(T_i)+1$. The bit $b(T_i)$, defined below, distinguishes between these two possibilities.

$$b(T_i) = \begin{cases} 1, & \text{if } r(T_{i+1}) = d(T_i) \\ 0, & \text{if } r(T_{i+1}) = d(T_i) + 1. \end{cases}$$

As we will see shortly, in the event of a tie, PD favors a subtask deadline with a b-value of 1 over one with a b-value of 0. Informally, it is better to execute a subtask T_i "early" if its window overlaps that of its successor T_{i+1}. By not choosing to execute T_i "early" we may end up having to schedule it in the last slot of its window. In effect, this reduces the length of T_{i+1}'s window by one. Thus, it is desirable to tie-break a deadline with a b-value of 1 over a deadline with a b-value of 0 because this places fewer constraints on future time slots.

Group deadlines. 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. It can be shown that a task is heavy if and only if the first window of each of its jobs is of length two, and all of its windows are of length two or three (this property is proved in [1]). Because heavy tasks have such "small" windows, they are more difficult to schedule correctly than light tasks. To see this, consider a sequence $T_i, \ldots, T_j$ of subtasks of a heavy task T such that $|w(T_k)| = 2 \wedge b(T_k) = 1$ for all $i < k \leq j$ and either $|w(T_{j+1})| = 3$ or $b(T_j) = 0$ (e.g., T_i, T_{i+1} or T_{i+2}, T_{i+3}, T_{i+4} or T_{i+5}, T_{i+6}, T_{i+7} in Figure 1). If any one of the subtasks $T_i, \ldots, T_j$ is scheduled in the last slot of its window, then each subsequent subtask in this sequence must be scheduled in its last slot. In effect, $T_i, \ldots, T_j$ must be considered as a single schedulable entity subject to a "group" deadline. Formally, we define the *group deadline* for the group of subtasks $T_i, \ldots, T_j$ to be $d(T_j) + 1$ if $|w(T_{j+1})| = 3$, and $d(T_j)$ if $b(T_j) = 0$. 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. For example, in Figure 1, T has group deadlines at slots $t + 3$, $t + 7$, and $t + 10$.

We let $D(T_i)$ denote the group deadline of subtask T_i. Formally, if T is heavy, then

$$D(T_i) = (\mathbf{min}\ u :: u \geq d(T_i) \text{ and } u \text{ is a group deadline of } T).$$

For example, in Figure 1, $D(T_i) = t + 3$ and $D(T_{i+5}) = t + 10$. The above definition of D is valid only for heavy tasks. If T is light, then $D(T_i) = 0$.

Given the above discussion of group deadlines, we are now in a position to state the PD priority definition.

PD Priority Definition: Task T's priority at time t is defined to be $(d(T_i), b(T_i), D(T_i))$, where T_i is eligible at time t. Priorities are ordered according to the following relation.

$$(d', b', D') \preceq (d, b, D) \equiv [d < d'] \vee [(d = d') \wedge (b > b')] \vee [(d = d') \wedge (b = b') \wedge (D \geq D')]$$

If T_i and U_j are eligible at time t, then T's priority is at least U's at time t if $(d(U_j), b(U_j), D(U_j)) \preceq (d(T_i), b(T_i), D(T_i))$. $\square$

According to this definition, task T has higher priority than task U if the pseudo-deadline of T's current subtask is less than that of U's. If two subtasks have the same pseudo-deadline, then the bit that we defined for each pseudo-deadline is used as a tie-breaker. If two heavy subtasks have equal pseudo-deadlines and the bits associated with them are the same, then the subtask with the greater group deadline has higher priority. Note that choosing *not* to schedule the subtask with the greater group deadline in this case would more tightly constrain scheduling decisions at future time slots. Due to the definition of D, if a heavy subtask and a light subtask have identical pseudo-deadlines and associated bits, then the tie is always resolved in favor of the heavy subtask. If a set of light-only tasks is to be scheduled, then the D parameter is not needed. Finally, note that it is possible for two tasks to have identical priorities; such ties can be broken arbitrarily.

Given this priority definition, PD is simple to explain. A priority-sorted "ready queue" is used to store eligible subtasks. In addition, as explained below, there are a number of priority-ordered "release queues" associated with future time slots. At the beginning of each time slot, the M highest-priority subtasks in the ready queue (if that many subtasks are eligible) are selected for execution, where M is the number of processors in the system. If T_i is one of the selected subtasks, then its successor T_{i+1} is inserted into the release queue Q_t, where t is the release time of T_{i+1}. At time t, Q_t will be merged with the ready queue. Thus, at the beginning of each time slot, the release queue for that slot is merged with the ready queue. An additional search structure is used in order to efficiently access the release queues.

ER-PD is a much simpler than PD, and is quite similar to more conventional priority-driven scheduling algorithms. With ER-PD, if T_i is selected for execution, and if its successor T_{i+1} is part of the same job, then T_{i+1} is inserted into the ready queue itself. If T_i and T_{i+1} are part of different jobs, then T_{i+1} is inserted into an appropriate release queue, as in PD. ER-PD is more efficient than PD because queue-merge operations are performed much less frequently than in PD.

4 Correctness Proof for ER-PD

We now prove that ER-PD produces an ERfair schedule. Throughout this section, we assume that $\sum_{T\in\tau}\frac{T.e}{T.p} = M$, where M is the number of processors. If less than M, then several dummy tasks can be added to make $\sum_{T\in\tau}\frac{T.e}{T.p} = M$. The proof proceeds by showing the existence of certain schedules over the interval $[0, L)$, where $L = LCM_{T\in\tau}(T.p)$. To facilitate our description of these schedules, we find it convenient to totally order all subtasks that are released in $[0, L)$. Let $\prec$ be an irreflexive total order that is consistent with the $\preceq$ relation in the PD priority definition. ($\prec$ is obtained by arbitrarily breaking any ties left by $\preceq$.) Let T_i and U_j be two subtasks that are released in $[0, L)$. Then, T_i is ordered before U_j, denoted $T_i \lhd U_j$ if and only if T_i has a higher PD priority than U_j when the issue of eligibility is ignored, i.e., $(d(U_j), b(U_j), D(U_j)) \prec (d(T_i), b(T_i), D(T_i))$. We let $\unlhd$ denote the reflexive counterpart of $\lhd$.

Our correctness proof for ER-PD uses several results from the proof given by us for PD in [1]. There, the correctness of PD is established by showing that if there exists a Pfair schedule S that is in accordance with PD up to time $t-1$, then such a schedule exists that is in accordance with PD up to time t. The number of scheduling decisions in S at time t that violate the PD priority definition can be inductively reduced to zero by swapping some subtasks. Such swappings will be valid if all subtasks are still scheduled within their Pfair windows, no two subtasks of the same task are scheduled in the same time slot, and M tasks are scheduled in each time slot. The crux of the proof lies in showing that if, in schedule S, **(i)** U_j is scheduled in slot t, **(ii)** T_i is eligible at t but scheduled later, and **(iii)** $T_i \lhd U_j$, then T_i can be swapped into slot t, and U_j can be swapped to a later slot. This is illustrated in Figure 3(a). If U_{j+1} is not scheduled in the same slot as T_i, then T_i and U_j can be directly swapped to get the desired schedule. On the other hand, if U_{j+1} *is* scheduled in the same slot as T_i, then U_j and T_i must have the same deadline and the windows of U_j and U_{j+1} must overlap by one slot, as shown in Figure 3(b). In this case, a swapping that is more complicated than a direct exchange of U_j and T_i is required. It turns out that a rather large number of cases must be considered to show that a valid swapping always exists. Fortunately for us, all of the swappings considered in [1] continue to be valid if, instead of requiring the entire schedule S to be Pfair, we only require Pfairness for the set of subtasks $\{V_k \mid T_i \unlhd V_k\}$ (only subtasks not in this set can be scheduled before the Pfair windows in S). This fact allows

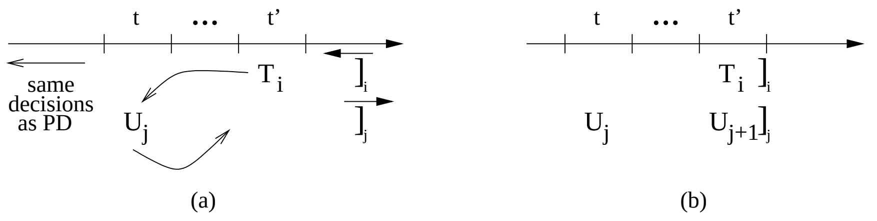

Figure 3: We use the following notation in this and subsequent figures. "[" and "]" indicate the release and deadline of a subtask; subscripts indicate which subtask. Each task is shown on a separate line. An arrow from a subtask V_k to a slot u indicates that V_k is now scheduled in slot u. An arrow over "[" (or "]") indicates that the actual position of "[" (or "]") can be anywhere in the direction of the arrow. Time is divided into unit-time slots that are numbered. If T_i is released at slot t, then "[" is aligned with the left side of slot t. If T_i has a deadline at slot t, then "]" is aligned with the right side of slot t. **(a)** Induction step of the PD correctness proof. **(b)** The "difficult" case to consider.

us to encapsulate all of the reasoning in [1] into the following lemma.

Lemma 1 *Let T_i and U_j be two subtasks in the ERfair schedule S, where*

- *U_j is scheduled in slot t,*
- *t lies within T_i's window, but T_i is scheduled at a later slot,*
- *$T_i \lhd U_j$, and*
- *for each V_k such that $T_i \unlhd V_k$, V_k is scheduled within its Pfair window.*

Then, there exists a schedule S' such that

- *S' and S are identical prior to slot t,*
- *$S'_t = S_t \cup \{T_i\} - \{U_j\}$.*
- *at each slot after t in S', M subtasks from M distinct tasks are scheduled, and*
- *each subtask V_k such that $T_i \unlhd V_k$ is scheduled in its Pfair window.*

The situation addressed in Lemma 1 is like that depicted in Figure 3(a), except that only T_i and subtasks ordered after T_i by $\lhd$ are required to execute within their Pfair windows. Subtasks ordered earlier than T_i by $\lhd$ can be executed prior to their Pfair windows; such a subtask must executed within a slot where it is considered to be eligible under ERfair scheduling. The lemma states that it is possible to move U_j to the right out of slot t, and T_i to the left into slot t. As stated earlier, the swappings that must be considered to show that this is always possible involve only T_i and subtasks ordered after T_i by $\lhd$. Slot t is assumed to lie within T_i's Pfair window, so the resulting schedule will still be Pfair for these subtasks.

To facilitate our proof of correctness for ER-PD, we define an ERfair schedule S to be *k-compliant* if and only if **(i)** each subtask that is not among the first k according to the relation $\lhd$ is scheduled within its Pfair

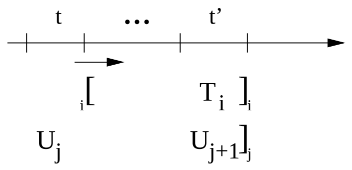

Figure 4: The "difficult" case to consider in the proof of Lemma 2.

window, and **(ii)** the first k subtasks according to $\lhd$ are scheduled in accordance with ER-PD. We now prove that a k-compliant schedule exists by induction on k. Note that a 0-compliant schedule is just a Pfair schedule, and the existence of such a schedule follows from [1]. Also, if N is the total number of subtasks in $[0, L)$, then an N-compliant schedule is an ERfair schedule that is fully in accordance with ER-PD. The following lemma gives the inductive step of the proof.

Lemma 2 *If S is a k-compliant schedule, then there exists a schedule S' that is $(k+1)$-compliant.*

Proof: Let T_i be the $(k+1)^{st}$ subtask according to $\lhd$. If T_i is scheduled in S in accordance with ER-PD, then take S' to be S. Otherwise, we have the following: there exists a time slot t such that T_i is eligible at t, some subtask ordered after T_i by $\lhd$ is scheduled at t, and T_i is scheduled later than t. Take t to be the earliest such time slot, and let U_j be the lowest-priority subtask scheduled at t. Let t' be the slot where T_i is scheduled. If t lies within T_i's Pfair window, then we can apply Lemma 1 to get the desired schedule S'. In the remainder of the proof, assume that t is before T_i's Pfair window. Note that if U_{j+1} is not scheduled in slot t', then we can directly swap T_i and U_j to get S'. In the rest of the proof, we assume U_{j+1} *is* scheduled in slot t'. As discussed above, this implies that U_j and T_i have the same deadline and the Pfair windows of U_j and U_{j+1} overlap by one slot. This situation is depicted in Figure 4. We now prove that a valid swapping exists for this remaining situation by considering four cases, which depend on the weights of tasks T and U. In these four cases, We make use of the following properties concerning Pfair windows, which are proved in [1].

(P1) The length of each of task T's windows is either $\left\lceil \frac{T.p}{T.e} \right\rceil$ or $\left\lceil \frac{T.p}{T.e} \right\rceil + 1$. A window of task T with length $\left\lceil \frac{T.p}{T.e} \right\rceil$ (respectively, $\left\lceil \frac{T.p}{T.e} \right\rceil$+1) is called a *minimal* (respectively, *maximal*) window of T.

(P2) A task has a 2-window if and only if it is heavy.

(P3) If two successive subtasks T_i and T_{i+1} have windows that do not overlap, then we call them *nonoverlapping* subtasks (as a special case, T_1 is defined to be nonoverlapping as well). The window of each nonoverlapping subtask is a minimal window.

All of swappings in the cases that follow involve only T_i and subtasks ordered after T_i by $\lhd$. We remind the reader that, by assumption, all such subtasks are scheduled within their Pfair windows in S.

Case 1: *T* is light and *U* is heavy. By the definition of D, T cannot have higher priority than U at time t.

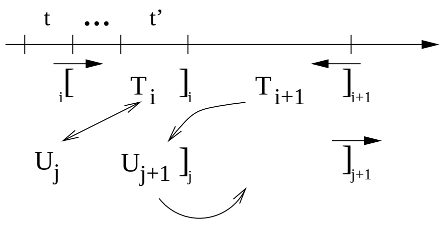

Figure 5: Case 2. T is heavy, U is light, and $d(T_i) = d(U_j)$.

Case 2: *T* is heavy and *U* is light. In this case, we show that the swapping in Figure 5 is valid. Because T_i has higher priority than U_j, the b-bit associated with T_i's deadline is not 0, i.e., T_{i+1}'s Pfair window begins in slot t'. As seen in Figure 5, U_{j+1}'s window also begins in slot t'. Because T is heavy and U is light, by (P1), each of T's windows is of length two or three, and each of U's windows is of length three or greater. Thus, $d(T_{i+1}) \leq d(U_{j+1})$. If T_{i+1} is a nonoverlapping subtask, then by (P2) and (P3), its window is of length two, and hence $d(T_{i+1}) < d(U_{j+1})$. Thus, we have $(d(T_{i+1}) < d(U_{j+1})) \lor (d(T_{i+1}) \leq d(U_{j+1}) \land b(T_{i+1}) = 1)$. Therefore, by the PD priority definition, $T_{i+1} \lhd U_{j+1}$. Hence, by Lemma 1, there exists a swapping that moves T_{i+1} into slot t' and U_{j+1} to a later slot. This implies that the swapping in Figure 5 is valid.

Case 3: *T* and *U* are both light. In this case, we show that one of the swappings in Figure 6 is valid. Because U is light, by (P2), $|w(U_{j+1})| \geq 3$. Therefore, U_{j+2} is released after $t' + 1$ (refer to Figure 3(b)), and hence no subtask of U is scheduled in slot $t' + 1$. If T_{i+1} is scheduled in slot $t' + 1$, then the swapping in Figure 6(a) is valid. In the rest of the proof for Case 3, we assume that T_{i+1} is scheduled after slot $t' + 1$.

Because $U \in S_{t'}$ and $U \notin S_{t'+1}$, and because all processors are fully utilized, there exists a task V such that $V \notin S_{t'}$ and $V \in S_{t'+1}$. Let V_k be the subtask of V scheduled in slot $t' + 1$. Because V_k is scheduled at time $t' + 1$, we have $r(V_k) \leq t' + 1$. If $r(V_k) < t' + 1$, then the swapping shown in Figure 6(b) produces the desired schedule. In the rest of the proof for Case 3, we assume $r(V_k) = t' + 1$, in which case this swapping is not valid.

Consider subtask V_{k-1}. If V_{k-1} is scheduled in the interval (t, t'), then the swapping shown in Figure 6(c) is valid. (Note that V_{k-1} either has a deadline at $t' + 1$ or has a deadline at t' and a b-bit of 0; hence, V_{k-1} is ordered after T_i by $\lhd$.) On the other hand, if V_{k-1} is scheduled in slot t, then we have a contradiction of our choice of U_j as the lowest-priority subtask scheduled in slot t (because V_{k-1} either has a deadline at $t' + 1$ or a deadline at t' and a b-bit of 0, it has lower priority than U_j). The remaining possibility is that V_{k-1} is scheduled before t. As shown in the following paragraph, $d(T_{i+1}) < d(V_k)$ holds in this case. This implies that T_{i+1} and V_{k+1} cannot be scheduled in the same slot, and hence, the swapping shown in Figure 6(d) is valid.

To facilitate the proof that $d(T_{i+1}) < d(V_k)$, we let $c = |w(V_{k-1})|$ and let $d = |w(T_i)|$. As seen in Figure 6(d), $c > d$. V_{k-1}'s deadline must be either at time t' or at time $t' + 1$. First, suppose it is at time t'. In this case, V_{k-1} and V_k are nonoverlapping subtasks, and hence by (P3), $|w(V_k)| = c$. Furthermore, by (P1), we have $|w(T_{i+1})| \leq d + 1$, which implies that $|w(T_{i+1})| \leq |w(V_k)|$. Because $w(V_k)$ begins one slot later than $w(T_{i+1})$,

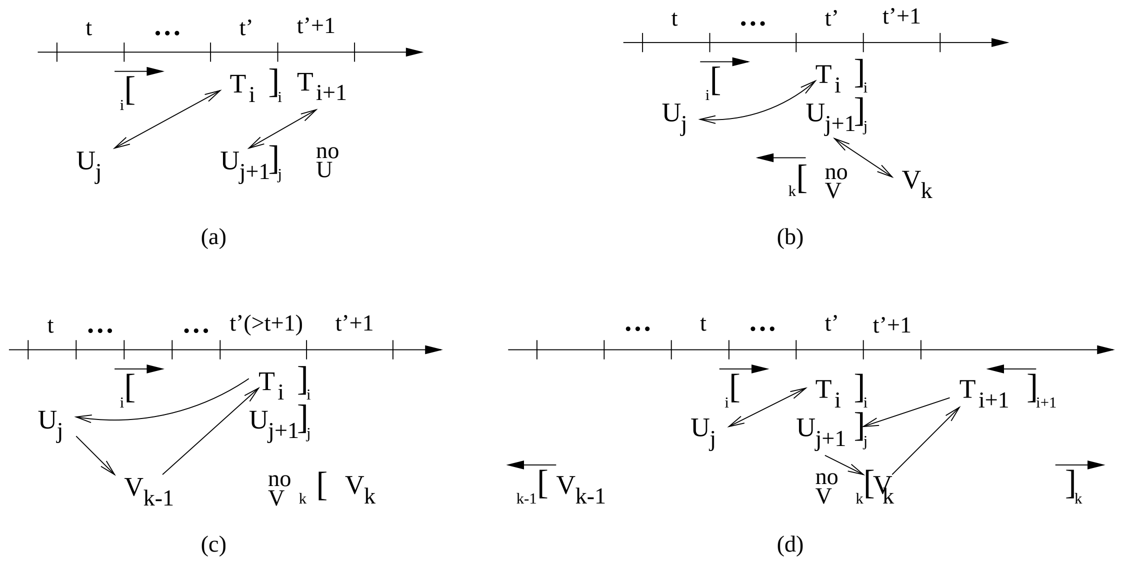

Figure 6: Case 3. **(a)** T_{i+1} is scheduled in slot $t' + 1$. **(b)** $r(V_k) < t' + 1$. **(c)** $r(V_k) = t' + 1$, and V_{k-1} is scheduled in the interval (t, t'). **(c)** $r(V_k) = t' + 1$, and V_{k-1} is scheduled before slot t.

this implies that $d(T_{i+1}) < r(V_k)$, as claimed. The other case to consider is that V_{k-1}'s deadline is at time $t'+1$. In this case, as seen in Figure 6(d), $d \leq c - 2$. By (P1), we have $|w(V_k)| \geq c - 1$ and $|w(T_{i+1})| \leq d + 1$. So, once again, we have $|w(T_{i+1})| \leq |w(V_k)|$, which implies that $d(T_{i+1}) < r(V_k)$. This completes the proof for Case 3.

Case 4: *T* and *U* are both heavy. In this case, by (P1) and (P2), t' in Figure 4 must be $t + 2$, i.e., the situation to consider is as depicted in Figure 7(a). As explained in Case 2, T_{i+1} and U_{j+1} both have windows starting at slot $t + 2$. Moreover, because T and U are both heavy, by (P1) and (P2), each of $w(T_{i+1})$ and $w(U_{j+1})$ is either a 2-window or a 3-window. If $w(T_{i+1})$ is a 2-window, then $T_{i+1} \lhd U_{j+1}$. To see this, note that if $w(U_{j+1})$ is of length three, then T_{i+1} has an earlier deadline than U_{j+1}; if $w(U_{j+1})$ is of length two, then T_{i+1} and U_{j+1} have equal deadlines, but the PD tie-break favors T_{i+1} over U_{j+1} since it favored T_i over U_j. Because $T_{i+1} \lhd U_{j+1}$, by Lemma 1, there exists a swapping that moves T_{i+1} into slot $t + 2$ and U_{j+1} to a later slot. Thus, the swapping in Figure 7(b) is valid.

Now, suppose that $w(T_{i+1})$ is a 3-window. This implies that T has a group deadline at time slot $t+3$. Because the PD tie-break favored T_i over U_j, this implies that $w(U_{j+1})$ is either a 3-window or a nonoverlapping 2-window (if $w(U_{j+1})$ were a 2-window overlapping $w(U_{j+2})$, then U_j's group deadline would be after $t + 3$, and hence, the PD tie-break would favor U_j over T_i). Because $w(U_{j+1})$ is either a 3-window or a nonoverlapping 2-window, and because U_{j+1} is scheduled in slot $t + 2$ (the first slot of its window), no subtask of U is scheduled in slot $t + 3$. Thus, if T_{i+1} is scheduled in slot $t + 3$, then the swapping in Figure 7(c) is valid.

In the rest of the proof for Case 4, we assume that T_{i+1} is *not* scheduled in slot $t + 3$, which implies that it must be scheduled in slot $t + 4$. Because $U \in S_{t+2}$ and $U \notin S_{t+3}$, and because all processors are fully utilized, there exists a task V such that $V \notin S_{t+2}$ and $V \in S_{t+3}$. Let V_k be the subtask of V scheduled in slot $t + 3$. If V_k is released at or before slot $t + 2$, then the swapping in Figure 7(d) is valid. Also, if V_k has a deadline

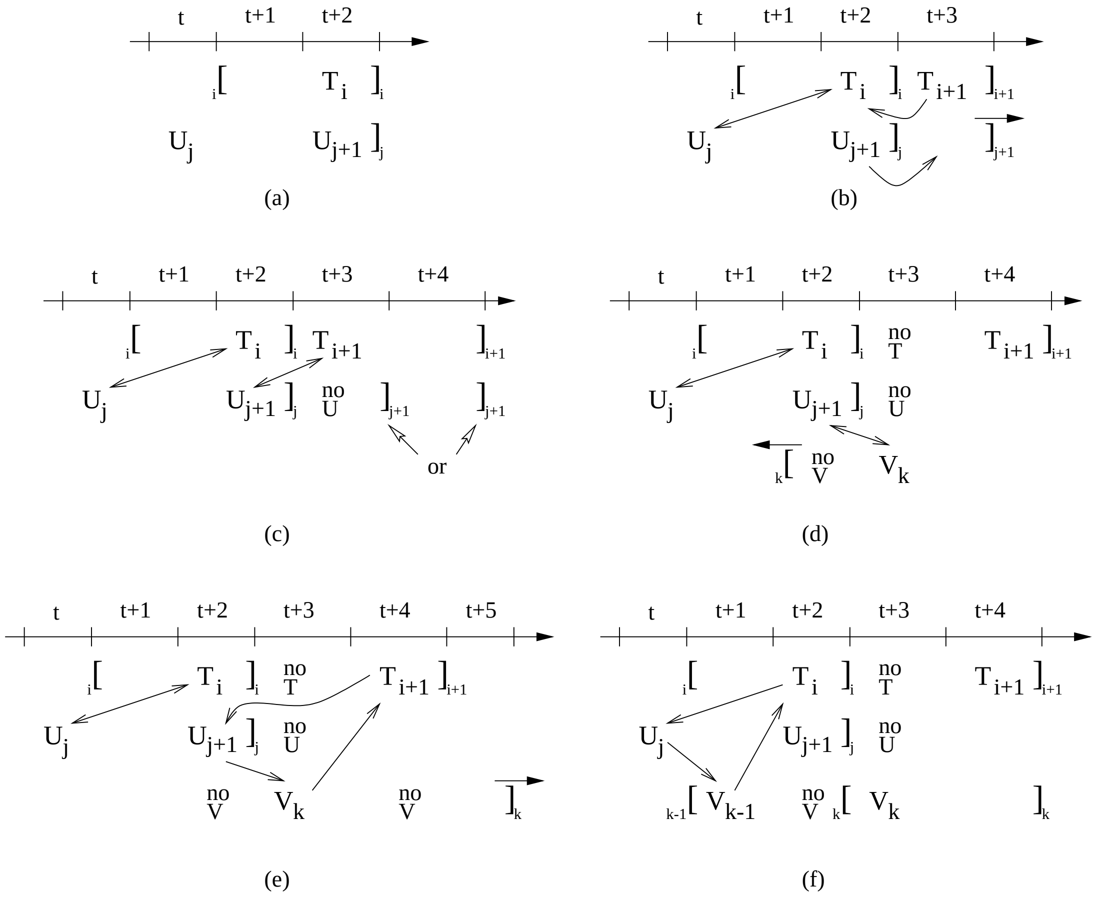

Figure 7: Case 4. **(a)** The situation to consider if T and U are both heavy. **(b)** $w(T_{i+1})$ is a 2-window. **(c)** $w(T_{i+1})$ is a 3-window, and T_{i+1} is scheduled in slot $t+3$. **(d)** $w(T_{i+1})$ is a 3-window, T_{i+1} is scheduled in slot $t+4$, and $r(V_k) \leq t+2$. **(e)** $w(T_{i+1})$ is a 3-window, T_{i+1} is scheduled in slot $t+4$, and $d(V_k) \geq t+5$. **(f)** $w(T_{i+1})$ is a 3-window, T_{i+1} is scheduled in slot $t+4$, and $w(V_k) = [t+3, t+4]$.

after slot $t+4$, then no subtask of V is scheduled in slot $t+4$, and hence the swapping in Figure 7(e) is valid. The remaining possibility is that V_k is released at slot $t+3$ and has a deadline at slot $t+4$, which by (P2), implies that V is heavy. Because V is heavy, $w(V_{k-1})$ begins in slot $t+1$ (it is either a nonoverlapping 2-window $[t+1, t+2]$ or a 3-window $[t+1, t+3]$). Moreover, because $V \notin S_{t+2}$, V_{k-1} is scheduled in slot $t+1$. This implies that the swapping in Figure 7(f) is valid. (Note that V_{k-1} either has a deadline at $t'+1$ or has a deadline at t' and a b-bit of 0; hence, V_{k-1} is ordered after T_i by $\lhd$.) This completes the proof for Case 4. □

By applying Lemma 2 inductively as discussed above, there exists an ERfair schedule consistent with ER-PD. Thus, we have the following theorem.

Theorem 1 *A task system τ can be scheduled correctly in an ERfair system using our priority definition if and only if $\sum_{T \in \tau}(T.e/T.p) \leq M$, where M is the number of processors.*

Lemma 2 actually allows us to conclude a much more general result than that stated in Theorem 1. In essence, the lemma shows that any subtask that is eligible before its Pfair window can be properly scheduled using the PD priority definition. There is much freedom here in defining when a subtask is eligible. In particular, there is no requirement that "early" eligibility be an option for *all* subtasks or even all subtasks of the *same* task. For that matter, a subtask's eligibility can be restricted so that it can become eligible only up to a fixed number of slots before its window. The only crucial requirement is this: if a subtask T_i is eligible at a slot t prior to $w(T_i)$, then T_i and T_{i-1} must be part of the same job, and T_{i-1} must be scheduled before t.

5 Experimental Results

In this section, we discuss the results of some preliminary experiments we conducted to compare the runtime overheads of ER-PD and PD. The runtime costs of the two algorithm differ only in the number of queue-merge operations performed. In particular, consider the interval $[0, L)$, where L is the least common multiple of all task periods. Let n_{jobs} and $n_{subtasks}$ denote the total number of jobs and subtasks, respectively, released in this interval. In both algorithms, each subtask must be enqueued onto some priority queue and eventually dequeued. Thus, each algorithm will perform $n_{subtasks}$ enqueue and dequeue operations in total. It should be noted that in most mergeable priority-queue algorithms, enqueue and dequeue operations are quite simple in comparison to queue-merge operations. Thus, we would expect the cost of queue-merge operations to be the dominate factor in the runtime overhead of both algorithms. In PD, a queue-merge operation must be performed at each slot where a subtask is released. The number of such slots is upper bounded by $n_{subtasks}$. In contrast, in ER-PD, a queue-merge operation must be performed at each slot where a *job* is released. The number of such slots is upper bounded by n_{jobs}.

In order to determine how many fewer queue-merge operations ER-PD performs than PD, we conducted some experiments involving randomly-generated task sets. The results of these experiments are shown in Figure 8. Four plots are shown in the figure corresponding to systems of 8, 16, 32, and 64 processors, respectively. In each plot, the x-axis represents the number of tasks in each randomly-generated task set. Task sets of 10 to 500 tasks were considered (in steps of 10). The y-axis gives the number of queue-merge operations performed up to time slot L (the least common multiple of all task periods). Each data point was obtained by averaging over 40 randomly-generated task sets. As these plots show, ER-PD performs about half as many queue-merge operations as PD. This is because the methodology we followed in generating task sets resulted in an average of two subtasks per job. (Note that determining the number of queue-merge operations for each algorithm is not simply a matter of analytically determining the number of jobs and subtasks released in $[0, L)$. In particular, we are interested in the number of *slots* at which jobs or subtasks are released, and there may be multiple releases per slot.) In general, ER-PD's gain over PD should roughly equal the number of subtasks per job averaged over all jobs in the system.

The experiments reported here obviously only provide a rough comparison of the two algorithms' runtime costs. In future work, we plan to empirically evaluate real implementations of both algorithms on a multiprocessor testbed.

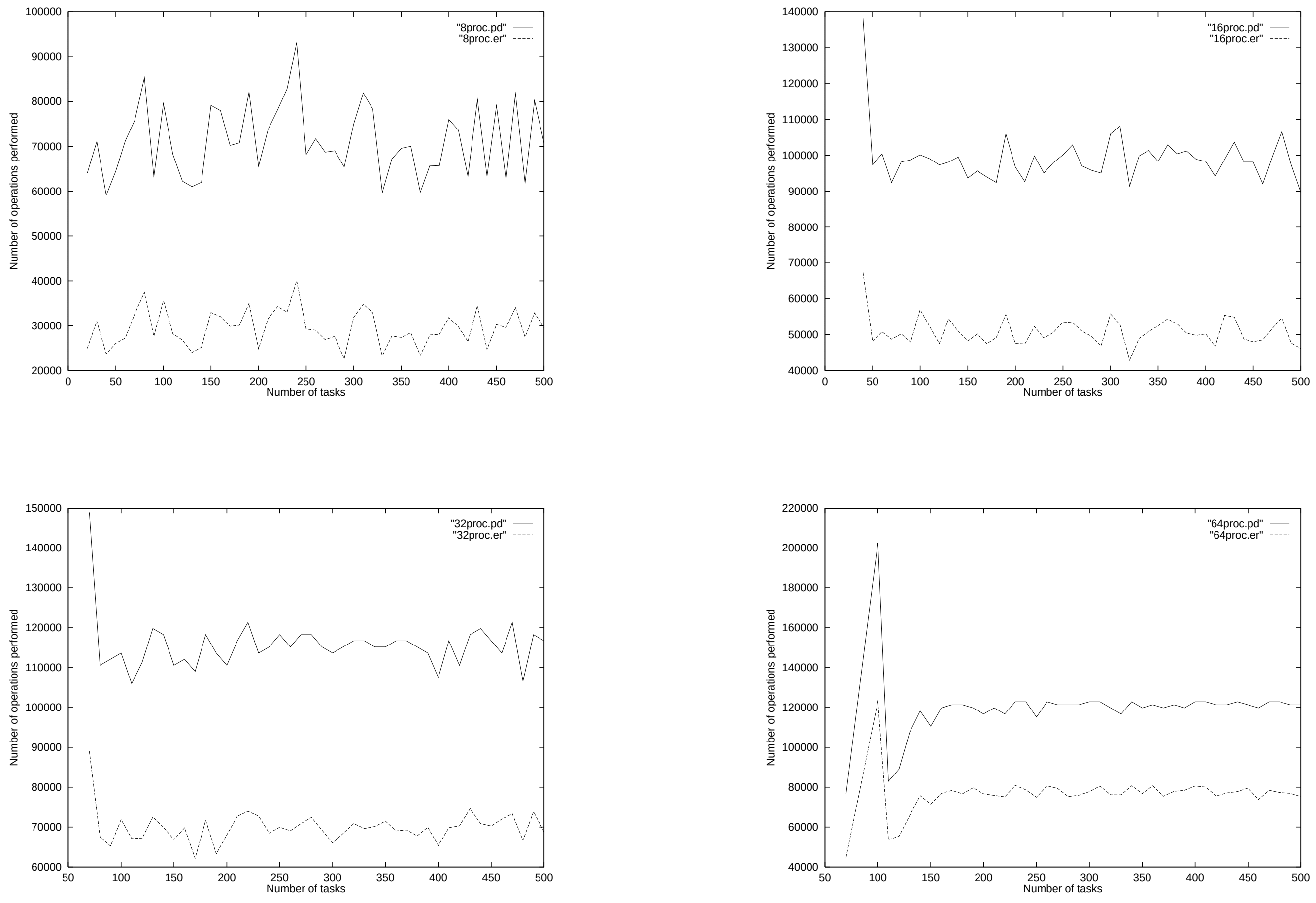

Figure 8: A comparison of the number of queue-merge operations performed by ER-PD and PD. The x-axis represents the number of tasks in the system. The y-axis gives the number of queue-merge operations performed up to the least common multiple of all task periods. Each data point was obtained by averaging over 40 randomly-generated task sets. Plots are shown for 8, 16, 32, and 64 processors. In all cases, ER-PD performs about half as many queue-merge operations as PD.

6 Concluding Remarks

We have presented a work-conserving variant of Pfair scheduling called early-release fair (ERfair) scheduling. Our notion of ERfair scheduling is obtained by simply dropping the -1 lag constraint used in Pfair scheduling. Without this constraint, subtasks can sometimes execute "early," i.e., before their Pfair windows. We have also presented an ERfair scheduling algorithm called ER-PD that can be used to efficiently schedule any task system whose total utilization is at most the number of available processors. ER-PD was obtained from the related PD algorithm for Pfair systems. We have shown that ER-PD entails lower runtime overhead than PD and that job response times can be much less with ER-PD than with PD.

It follows from our correctness proof for ER-PD that a variety of "hybrid" optimal schedulers exist that incorporate aspects of both PD and ER-PD. For example, it is possible to declare certain tasks to be "early releasable" and others not. This might be useful if a small subset of tasks in a system are subject to stringent average response-time requirements. It is also possible to dynamically decide when and by how much subtasks may be released early. Such flexibility might be useful in systems in which average response-time requirements must be balanced against jitter requirements. The development of a methodology for analyzing response-time/jitter tradeoffs is a subject of ongoing research.

In recent work, we have shown that PD and ER-PD are correct not only for periodic task systems, but also sporadic and "intra-sporadic" systems [2]. In an *intra-sporadic* task system, additional separation is allowed not only between jobs of the same task, as in a sporadic system, but also between the windows *within* a job. The intra-sporadic/early-release (IS/ER) model supports a very flexible notion of a rate without sacrificing optimality. This notion of a rate allows allocation errors to be quite arbitrary within a job release. However, such allocation errors are not allowed to bridge across job releases. In essence, each task's allocation error must periodically be reset to zero. When applied within the IS/ER model, ER-PD is a *multiprocessor* algorithm that functions in much the same way as the *uniprocessor* deadline scheduling algorithm proposed by Jeffay and Goodard for their rate-based execution (RBE) model [4]. Connections between the IS/ER and RBE models warrant further study.

Acknowledgement: We are grateful to Sanjoy Baruah and Kevin Jeffay for many helpful discussions concerning this paper.

References

[1] J. Anderson and A. Srinivasan. A new look at Pfair priorities. Submitted to *Real-time Systems Journal*, October 1999. Available at `http://www.cs.unc.edu/~anderson/papers.html`.

[2] J. Anderson and A. Srinivasan. Towards a more efficient and flexible Pfair scheduling framework. In *Proceedings of the Twentieth IEEE Real-Time Systems Symposium Work-in-progress Session*, pages 46–50, December 1999.

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

[4] K. Jeffay and S. Goddard. The rate-based execution model. In *Proceedings of the Twentieth IEEE Real-Time Systems Symposium*, pages 304–314, December 1999.

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