Quick-release Fair Scheduling*

James H. Anderson, Aaron Block, and Anand Srinivasan

Department of Computer Science

University of North Carolina at Chapel Hill

Abstract

In prior work on multiprocessor fairness, efficient techniques with provable properties for reallocating spare processing capacity have been elusive. In this paper, we address this shortcoming by proposing a new notion of multiprocessor fairness, called *quick-release fair* (*QRfair*) scheduling, which is a derivative of Pfair scheduling that allows efficient allocation of spare capacity. Under QRfair scheduling, each task is specified by giving both a minimum and a maximum weight (*i.e.*, processor share). The goal is to schedule each task (as the available spare capacity changes) at a rate that is **(i)** at least that implied by its minimum weight and **(ii)** at most that implied by its maximum weight. Our contributions are fourfold. First, we present a quick-release variant of the PD^2 Pfair scheduling algorithm called PD^{Q}. Second, we formally prove that the allocations of PD^{Q} always satisfy (i) and (ii). Third, we consider the problem of defining maximum weights in a way that encourages a fair distribution of spare capacity. Fourth, we present results from extensive simulation experiments that show the efficacy of PD^{Q} in allocating spare capacity.

Keywords: Dynamic systems, fair scheduling, multiprocessors, Pfairness, work-conserving scheduling.

*Work supported by NSF grants CCR 9972211, CCR 9988327, ITR 0082866, and CCR 0204312.

1 Introduction

There has been much recent work on scheduling techniques that ensure fairness, temporal isolation, and timeliness among tasks scheduled on the same resource. Much of this work is rooted in an idealized scheduling abstraction called *generalized processor sharing* (*GPS*). Under GPS scheduling, tasks are assigned weights, and each task is allocated a share of the resource in proportion to its weight. Thus, each task's designated share is guaranteed (fairness) and any "misbehaving" task is prevented from consuming more than its share (temporal isolation). In addition, real-time deadlines can be guaranteed, where feasible (timeliness).

GPS-fairness requires that, at all times, each task has been assigned precisely its required share of the resource thus far. In practice, this degree of fairness is impractical, as it requires the ability to preempt and swap tasks at arbitrarily small scales. Nonetheless, GPS serves as a useful "benchmark" against which practical algorithms can be judged: most such algorithms are designed to ensure that, over time, per-task allocations never deviate "too much" from GPS. The earliest work on GPS-like algorithms was directed at the problem of scheduling packets in networks (*e.g.*, [6, 10, 11, 13]); here the "tasks" to be scheduled are packet flows. More recently, GPS-like algorithms for processor scheduling have been proposed (*e.g.*, [1, 2, 3, 4, 5, 8, 9, 14, 15, 16]). Much of this work has been directed at multiprocessor systems. Such systems are the focus of this paper.

In work on fairness in networks, the need to consider *dynamic* behavior is fundamental, because the set of flows passing through a router changes with time. In work on fair processor scheduling, dynamic systems have only recently been considered [5, 15, 16]. The distinguishing characteristic of such a system is that tasks are allowed to *join* and *leave*. Moreover, the weights of existing tasks may change (thereby increasing or decreasing their shares). Because of such changes, spare processing capacity may become available that can be reallocated to other tasks. The development of techniques for efficiently doing this has been one of the most important goals in prior work on fairness in uniprocessors and networks. To this end, the concept of *virtual time* was proposed [6, 10, 11, 13, 16]. In essence, virtual time is a scaling factor that can be applied to shrink or expand the deadlines of tasks, in order to increase or decrease their shares, as the amount of spare processing capacity changes.

Unfortunately, in the multiprocessor case, techniques with provable properties for reallocating spare capacity have been elusive. Indeed, uniprocessor notions of virtual time are not straightforward to extend to this case. While it is beyond the scope of this paper to discuss this in much detail, a single scaling factor appears to be insufficient when scheduling multiple resources. Because of such difficulties, previous researchers have resorted to using heuristics to allocate spare capacity [8]. While such heuristics may work well in certain situations, their lack of formal properties makes them of questionable utility in systems where predictability is a concern.

In this paper, we propose a new notion of multiprocessor fairness, called *quick-release fair* (*QRfair*) scheduling,

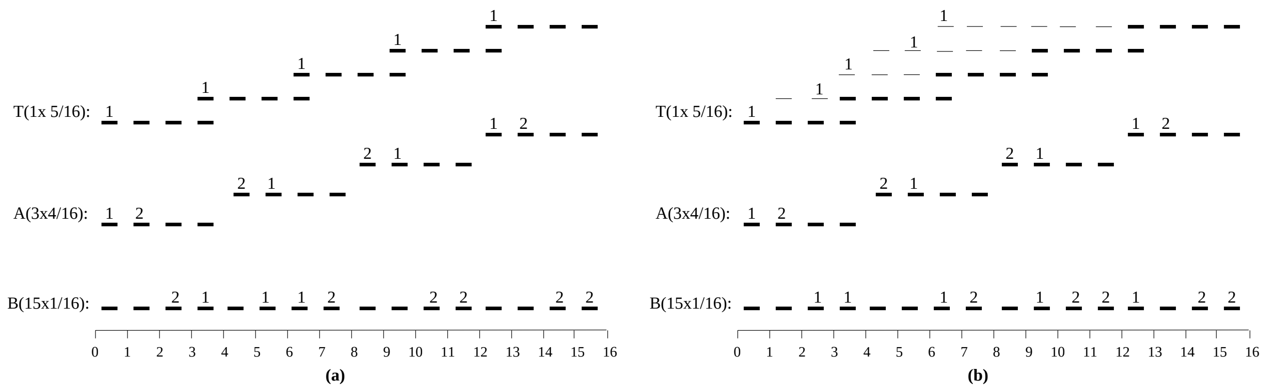


Figure 1: Two partial schedules on two processors are shown. In each schedule, tasks of a given weight are shown together. Each quantum-length subtask has an *eligibility interval*, denoted by dashes, corresponding to the sequence of time slots (*i.e.*, quanta) in which it can be scheduled; a subtask's eligibility interval includes its Pfair window, denoted in bold. An integer value n in slot t of some window means that n of the subtasks that must execute within that window are scheduled in slot t. No integer value means that no such subtask is scheduled in slot t. **(a)** All tasks are Pfair-scheduled. **(b)** Task T is ERfair-scheduled and all other tasks are Pfair-scheduled.

which has been devised with the goal of allocating spare processing capacity more seamlessly in multiprocessor systems. We also present an efficient QRfair scheduling algorithm and formally establish a number of properties concerning the allocation decisions it makes. The notion of QRfair scheduling is derived from earlier work on *proportionate fair* (*Pfair*) scheduling [3]. Before describing our main contributions in detail, we first present a brief overview of notions relevant to Pfair scheduling.

Pfair scheduling. Under Pfair scheduling [2, 3, 4], each task is required to execute at a uniform rate, while respecting a fixed allocation quantum. Each task's rate is specified by a rational *weight*, which gives its required utilization. Uniform rates are ensured by requiring the allocation error for each task to be always less than one quantum, where "error" is determined by comparing to an ideal GPS-like system. Due to this requirement, each task is effectively subdivided into quantum-length *subtasks* that must execute within *windows* of approximately equal lengths: if a subtask of a task T executes outside of its window, then T's error bounds are exceeded. The end of a subtask's window defines a *pseudo-deadline* for that task. A task's subtasks may execute on any processor, *i.e.*, migration is allowed. An example of a Pfair schedule is shown in Fig. 1(a). (The other inset of this figure is considered below.) The depicted schedule includes a task T of weight 5/16, a set A of three tasks of weight 4/16 each, and a set B of 15 tasks of weight 1/16 each, executing on two processors.

Under Pfair scheduling, if some subtask of a task executes "early" within its window, then it is ineligible for execution until the beginning of its next window. Thus, Pfair scheduling algorithms are not work conserving when used to schedule periodic tasks. Informally, a scheduling algorithm is *work conserving* if no processor ever idles unnecessarily. In [1], Anderson and Srinivasan proposed a work-conserving variant of Pfair scheduling, called *early-release fair* (*ERfair*) scheduling, which differs from Pfair scheduling in a rather simple way: under ERfair

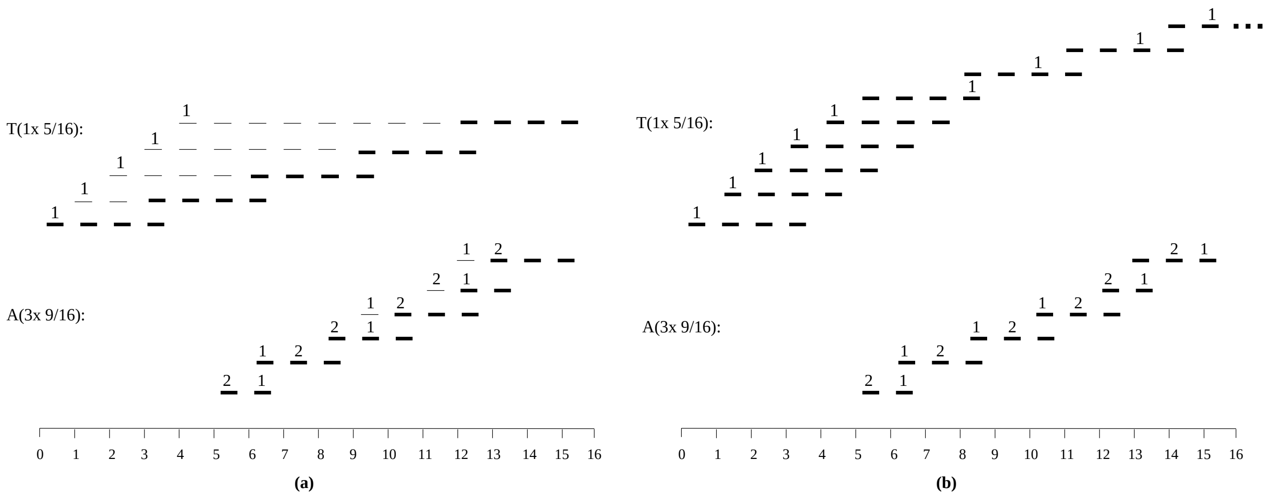


Figure 2: Partial schedules showing how spare capacity is utilized under **(a)** ERfair and **(b)** QRfair scheduling.

scheduling, a subtask may become eligible for execution *early*, *i.e.*, before its Pfair window. By allowing early releases, response times can often be reduced. This is illustrated in Fig. 1(b); note that T completes six time units earlier here than an inset (a). (Other task models that allow *late* releases are considered later in Sec. 2.)

Three algorithms have been devised that are optimal under Pfair and ERfair scheduling: PF [3], PD [4], and PD^2 [1]. These algorithms prioritize subtasks on an earliest-pseudo-deadline-first (EPDF) basis, but differ in the choice of tie-breaking rules. PD^2 is the most efficient of the three and uses two tie-break parameters.

Quick-release fair scheduling. To motivate the problem of allocating spare capacity in a multiprocessor system, consider the two-processor schedules in Fig. 2. The depicted system consists of one task T of weight $5/16$, which is present at time 0, and a set A of three tasks of weight $9/16$, which join the system at time 5. Note that spare capacity exists prior to time 5, but the system is fully utilized afterwards. In inset (a), task T makes use of the spare capacity by early releasing its first five subtasks. These subtasks "use up" the first five subtask deadlines of T. As a result, when the other tasks join the system at time 5, T is prevented from executing for a very long time. That is, the system treats T *unfairly* by penalizing it for having used spare capacity in the past. Such a scenario is precisely the kind of behavior uniprocessor notions of virtual time were devised to prevent.

Fig. 2(b) depicts the same scenario under QRfair scheduling. In comparing the two insets, it can be seen that the main difference is that task T's first five windows have been left shifted in inset (b). As a result, the first five subtask deadlines that T "uses up" do not correspond to deadlines far into the future. In fact, in the depicted scenario, T begins releasing subtasks at time 5 as if it had never used any spare capacity in the past.

The distinguishing characteristic of QRfair scheduling is as follows: if a processor is idle at time t, then each task's next subtask window can begin either at time $t+1$ or time $t+2$, if it otherwise would begin later. (Later, we indicate which windows may begin at $t+1$ and which at $t+2$.) Moreover, such subtasks are allowed to be released early, *i.e.*, before their windows. This allows the idle capacity in slot t to be utilized. We call such releases *quick releases*. Later, we show that the optimality of PD^2 is not compromised if quick releases are allowed.

When a task performs a quick release, the position of its next subtask window is left-shifted as much as possible. This allows it to execute at a rate that is greater than that implied by its Pfair weight. Note, however, that in an actual system, there may be an upper bound on the amount of capacity a given task can consume. If such a task's subtasks are always left-shifted as aggressively as possible, then it may be allocated more processor time than it can use. This is clearly no better than allowing processors to idle. For this reason, we consider a task model in which each task has both a *minimum* and *maximum* weight. A task's minimum weight corresponds to its Pfair weight. Its maximum weight controls the extent to which its quick-released subtasks can be left-shifted.

Contributions. There are four main contributions of this paper beyond introducing the concept of QRfair scheduling. First, we present a quick-release variant of PD^2 called PD^Q. PD^Q has the same asymptotic time complexity as PD^2 but supports both minimum and maximum task weights. To the best of our knowledge, PD^Q is the first fair multiprocessor scheduling algorithm that allows nontrivial maximum weights less than unity to be specified. Second, we *formally* prove that PD^Q always executes each task at a rate that is **(i)** at least that implied by its minimum weight and **(ii)** at most that implied by its maximum weight. Third, we consider the problem of defining maximum weights in a way that encourages a fair distribution of spare capacity. Fourth, we present results from extensive simulation experiments that show the efficacy of PD^Q in allocating spare capacity. In most of these experiments, curves plotted to compare PD^Q and ideal allocations are almost indistinguishable.

The rest of the paper is organized as follows. In Sec. 2, needed definitions are presented. In Secs. 3 and 4, PD^Q is presented and Property (i) mentioned above is proved. Issues related to defining and maintaining maximum weights are considered in Sec. 5. Simulation results are then given in Sec. 6. We conclude in Sec. 7.

2 Preliminaries

In this section, definitions relating to Pfair scheduling and the task models considered in this paper are given.

2.1 Pfair and ERfair Scheduling

In defining notions relevant to Pfair scheduling, we limit attention (for now) to periodic tasks.[1] A periodic task T with an integer *period* $T.p$ and an integer *execution cost* $T.e$ has a *weight* $wt(T) = T.e/T.p$, where $0 < wt(T) \leq 1$. A task T is *light* if $wt(T) < 1/2$, and *heavy* otherwise.

[1]Unless specified otherwise, we assume that each periodic task begins execution at time 0.

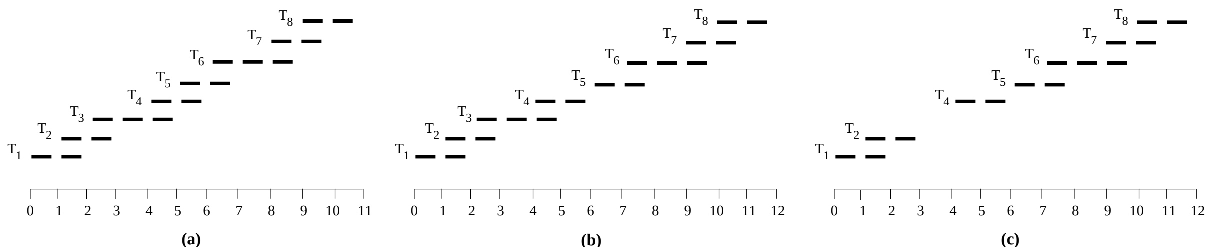


Figure 3: **(a)** Windows of the first job of a periodic task T with weight 8/11. This job consists of subtasks $T_1, \ldots, T_8$. Each of these subtasks must be scheduled during its window, or a lag-bound violation will result. (This pattern repeats for every job.) **(b)** The Pfair windows of an IS task. Subtask T_5 becomes eligible one time unit late. **(c)** The Pfair windows of a GIS task. Subtask T_3 is absent and T_5 is one time unit late. (Because T_3 is absent, this is not an IS task.)

Under Pfair scheduling, processor time is allocated in discrete time units, called *quanta*; the time interval $[t, t+1)$, where t is a nonnegative integer, is called *slot* t. (Hence, time t refers to the beginning of slot t.) The sequence of allocation decisions over time defines a *schedule*. Formally, a schedule S is a mapping $S : \tau \times \mathcal{N} \mapsto \{0, 1\}$, where τ is a set of tasks and $\mathcal{N}$ is the set of nonnegative integers; $S(T, t) = 1$ iff T is scheduled in slot t.

The notion of a Pfair schedule is defined by comparing to an ideal schedule that allocates $wt(T)$ processor time to task T in each slot. Deviance from the ideal schedule is formally captured by the concept of *lag*. The *lag of task T at time t*, $lag(T, t)$, is defined as $wt(T) \cdot t - \sum_{u=0}^{t-1} S(T, u)$. A schedule is *Pfair* iff $(\forall T, t :: -1 < lag(T, t) < 1)$. Informally, each task's allocation error must always be less than one quantum.

Each quantum of a task's execution, henceforth called a *subtask*, must be allocated without violating the lag bounds above. We denote the i^{th} subtask (*i.e.*, i^{th} quantum of allocation) of task T as T_i, where $i \geq 1$. Associated with subtask T_i is a *pseudo-release* $r(T_i)$ and *pseudo-deadline* $d(T_i)$ defined as follows.

$$\left(r(T_i) = \left\lfloor \frac{i-1}{wt(T)} \right\rfloor \right) \quad \wedge \quad \left(d(T_i) = \left\lceil \frac{i}{wt(T)} \right\rceil \right) \tag{1}$$

(For brevity, we often drop the prefix "pseudo-.") T_i must be scheduled in the interval $w(T_i) = [r(T_i), d(T_i))$, termed its *window*, or else a lag-bound violation will result. As an example, consider subtask T_2 in Fig. 3(a). Here, we have $r(T_2) = 1$, $d(T_2) = 3$, and $w(T_2) = [1, 3)$. Therefore, T_2 must be scheduled in either slot 1 or 2. (If T_1 is scheduled in slot 1, then T_2 must be scheduled in slot 2.)

ERfair scheduling. The notion of ERfair scheduling [1], mentioned earlier, is obtained by dropping the -1 constraint in the Pfair lag bounds above. With this change, a subtask can become eligible before its Pfair window.

2.2 The Intra-sporadic and Generalized Intra-sporadic Models

The intra-sporadic (IS) task model generalizes the well-known sporadic task model [12] by allowing subtasks to be released late [14]. This extra flexibility is useful in many applications where processing steps may be delayed (*e.g.*, due to jitter). Fig. 3(b) illustrates the Pfair windows of an IS task. Each subtask T_i of an IS task has an *offset* $\theta(T_i)$ that gives the amount by which its release has been delayed. By (1),

$$\left(r(T_i) = \theta(T_i) + \left\lfloor \frac{i-1}{wt(T)} \right\rfloor\right) \quad \wedge \quad \left(d(T_i) = \theta(T_i) + \left\lceil \frac{i}{wt(T)} \right\rceil\right). \tag{2}$$

These offsets are constrained so that the separation between any pair of subtask releases by a task is at least the separation between those releases if the task were periodic. Formally, the offsets satisfy the following property.

$$k \geq i \Rightarrow \theta(T_k) \geq \theta(T_i) \tag{3}$$

Each subtask T_i has an additional parameter $e(T_i)$ that corresponds to the first time slot in which T_i is eligible to be scheduled. It is assumed that $e(T_i) \leq r(T_i)$ and $e(T_i) \leq e(T_{i+1})$ for all $i \geq 1$. Allowing $e(T_i)$ to be less than $r(T_i)$ is equivalent to allowing "early" subtask releases as in ERfair scheduling. The interval $[r(T_i), d(T_i))$ is called the *PF-window* of T_i, while the interval $[e(T_i), d(T_i))$ is called the *IS-window* of T_i. (We stress that when we refer to the *release time* of a subtask, we mean the beginning of its PF-window. Such a subtask may become eligible for execution before its release time. Note that a subtask's release time determines its deadline.)

The generalized intra-sporadic (GIS) model generalizes the IS model by allowing subtasks to be absent. That is, a GIS task is obtained by removing some subtasks from an IS task of equal weight. Fig. 3(c) shows an example. If a task T, after executing subtask T_i, releases subtask T_k, then T_k is called the *successor* of T_i and T_i is called the *predecessor* of T_k. For example, T_4 is T_2's successor in Fig. 3(c). Note that subtask indices are assigned to reflect the missing subtasks. For example, task T in Fig. 3(c) releases subtask T_4 after releasing T_2; T_3 is missing and $\theta(T_4) = 0$. Hence, the formulae for subtask release times and deadlines of a GIS task are as in (2).

2.3 Scheduling Algorithms

Three Pfair scheduling algorithms are known to be optimal for scheduling GIS tasks on an arbitrary number of processors: PF [3], PD [4], and PD^2 [1]. Each prioritizes subtasks on an earliest-pseudo-deadline-first (EPDF) basis, but they use different tie-breaks. PD^2, the most efficient of the three, uses the following two tie-breaks.

***b*-bit.** $b(T_i)$ is defined as $\left\lceil \frac{i}{wt(T)} \right\rceil - \left\lfloor \frac{i}{wt(T)} \right\rfloor$. In a periodic task system, $b(T_i)$ is 0 if T_i's window does not overlap T_{i+1}'s, and is 1 otherwise. For example, in Fig. 3(a), $b(T_i) = 1$ for $1 \leq i \leq 7$ and $b(T_8) = 0$.

Group deadline. This parameter is needed in systems containing tasks with windows of length two; a task T has such windows iff $1/2 \leq wt(T) < 1$. The *group deadline* of subtask T_i, denoted $D(T_i)$, is defined as follows.

$$D(T_i) = \theta(T_i) + \left\lceil \frac{\left\lceil \left\lceil \frac{i}{wt(T)} \right\rceil \times (1 - wt(T)) \right\rceil}{1 - wt(T)} \right\rceil$$

A heavy periodic task has group deadlines at the end of each slot that is contained only within a single window. For example, the task in Fig. 3(a) has group deadlines at times 4, 8, and 11, and $D(T_4) = 8$, $D(T_5) = 8$, and $D(T_7) = 11$. For a GIS task, the group deadline is defined similarly assuming that all future subtasks are present and are released as early as possible. For example, in Fig. 3(c), $D(T_4) = 8$, $D(T_5) = 9$, and $D(T_7) = 12$. If T is a light task, then $D(T_i)$ is defined to be zero.

Under PD^2, if two subtasks have the same deadline, then a subtask with a b-bit of 1 is favored over one with a b-bit of 0. If both subtasks have a b-bit of 1, then the one with the larger group deadline is favored. Any further ties are broken arbitrarily. (Refer to [1] for a more detailed description of PD^2.)

In [14], Srinivasan and Anderson proved that PD^2 is optimal for scheduling GIS task systems, *i.e.*, it correctly schedules any GIS task system on M processors if the sum of the weights all tasks is at most M.

2.4 Dynamic Task Systems

In recent work, Srinivasan and Anderson [15] derived the following sufficient conditions under which GIS tasks may dynamically join and leave a running system, without causing any missed deadlines under PD^2.

(J) *Join condition*: A task T can join at time t iff the sum of the weights of all tasks after joining is at most M, the number of processors (follows from the feasibility test).

(L) *Leave condition*: Let T_i denote the last-scheduled subtask of T. If T is light, then T can leave at time t iff either $t = d(T_i) \wedge b(T_i) = 0$ or $t > d(T_i)$ holds. If T is heavy, then T can leave at time t iff $t \geq D(T_i)$.

Theorem 1 ([15]) PD^2 *correctly schedules any dynamic GIS task system satisfying* (J) *and* (L).

As shown in [15], these conditions are tight: if a task is allowed to leave earlier than allowed by (L), then it can re-join immediately, effectively executing at a higher rate. Deadlines may be missed as a result.

3 QRfair Scheduling and the PD$^{\mathrm{Q}}$ Scheduling Algorithm

Quick-release fair (QRfair) scheduling improves upon ERfair scheduling by more intelligently using idle processor time. In a schedule S, if k processors are idle in slot t, then we say that there are k *holes* in S in slot t. Note that holes may exist because of late subtask releases, even if total utilization is M. When a slot with a hole is encountered, various subtasks may be "quick released" using rules specified below. These rules allow each subtask T_i to have a *minimum separation parameter* $x(T_i)$ that gives the minimum value allowed for $r(T_i) - r(T_j)$, where T_j is the predecessor of T_i. The reason for allowing this parameter will become apparent later in Sec. 5.

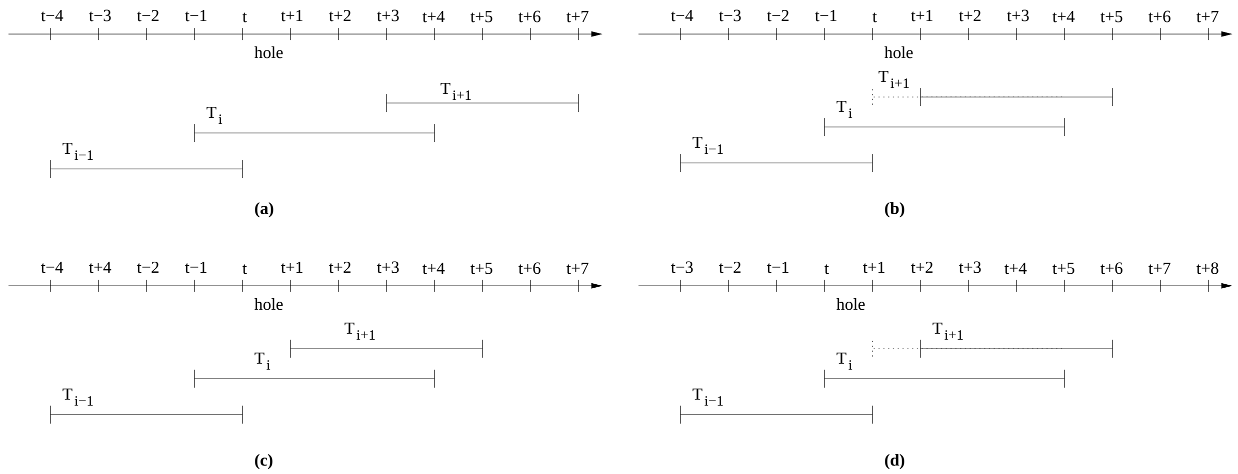


Figure 4: Illustration of the application of Rule A in a slot t that has a hole. Solid line segments depict PF-windows; dashed lines are used to show the extent to which an IS-window extends before a corresponding PF-window. **(a)** Pfair scheduling: Rule A is not applied. **(b)** T is $(t+1)$-releasable and T_i is scheduled before slot t. **(c)** T is $(t+1)$-releasable and T_i is scheduled in slot t. **(d)** T is $(t+2)$-releasable.

Quick-release rules. Let t be a slot with one or more holes. Let T_i be the last subtask of task T that is scheduled at or before t, and assume that T_i has a deadline after $t+1$. (If T has an earlier deadline, then its successor subtask's window may begin at t or $t+1$, so the rules given below have no impact.) We say that T is $(t+1)$-*releasable at time* t if it satisfies one of the following two properties: **(i)** T_i is scheduled before t; **(ii)** T_i is scheduled at t, its predecessor[2] subtask has a deadline at or before t, and T is light. Also, T is $(t+2)$-*releasable at time* t if T is light, T_i is scheduled in slot t, and its predecessor subtask has a deadline at time $t+1$.

Let T_k denote the successor subtask of T_i. By (2) and (3), $r(T_k) \geq t+1$. If $r(T_k) > t+1$, then under QRfair scheduling, we may modify the parameters of T_k using the following two rules. (It is not necessary to apply this modification for every such subtask.)

Rule A: If T is t'-releasable at time t (where t' is either $t+1$ or $t+2$), then $r(T_k)$ is modified to $max(t', r(T_i) + x(T))$. Thus, $\theta(T_k) = max(t', r(T_i)+x(T)) - \left\lfloor \frac{k-1}{wt(U)} \right\rfloor$, and $d(T_k) = max(t', r(T_i)+x(T)) - \left\lfloor \frac{k-1}{wt(U)} \right\rfloor + \left\lceil \frac{k}{wt(U)} \right\rceil$. $D(T_k)$ is modified in a similar manner but $b(T_k)$ remains unchanged.

Rule B: $e(T_k)$ may be assigned any value in $[t, r(T_k)]$.

Rule A tries to left-shift a subtask's PF-window (and hence, its deadline) as much as possible without violating the minimum separation between consecutive PF-windows. Fig. 4 illustrates this. The only case that Rule A does not cover is when a subtask T_i of a heavy task is scheduled in a slot with a hole. Since a heavy task has windows of length two or three [1], the window of T_i's successor subtask can begin at either $t+1$ or $t+2$ anyway.

[2]Note that if the predecessor subtask does not exist, then the conditions given here and in the next sentence are vacuously true. For brevity, we will avoid saying "if it exists" when considering these conditions later.

Rule B allows the shifted subtask to be eligible early. In our implementation of PD$^{\mathrm{Q}}$, we always set $e(T_k) := t$ when applying Rule B because this allows the idle capacity in slot t to be utilized. Rule B is stated in a more general way because this facilitates the correctness proof given later.

Once these rules are applied, T_i's predecessor, instead of T_i, is considered as T_k's predecessor in future applications of these rules. In other words, T_i is effectively considered as being absent. For example, in Fig. 4(c), T_{i-1} would now be considered as T_{i+1}'s predecessor. (The reason for this assumption will become clear in Sec. 4.)

Note that, since Rule A left-shifts windows, the resulting task system may no longer be a GIS system. In particular, the separation between consecutive windows of a task may be *less* than that for a periodic system.

The PD$^{\mathrm{Q}}$ algorithm. PD$^{\mathrm{Q}}$ is based on the earlier PD^2 algorithm and applies Rules A and B when it encounters slots with holes. PD$^{\mathrm{Q}}$ can be implemented in $O(M \log N)$ time, where M is the number of processors and N is the number of tasks. As in the implementation of PD and PD^2, task queues are implemented using binomial heaps. (The primary reason for using binomial heaps is that two such heaps can be merged in $O(\log n)$ time, where n is the total number of items in both heaps.) Pseudo-code for PD$^{\mathrm{Q}}$ is given in Fig. 5.

In Fig. 5, heap H contains all ready tasks. Heap Q contains subtasks of tasks that may be quick-released by Rule A. Heap J contains any new tasks that join the system. Heap $R[t]$ contains those subtasks that have PF-windows beginning at time t. The array S indicates the set of tasks that have been chosen for execution at time t, and n denotes the number of such tasks.

At any instant, if fewer than M subtasks are eligible, then the scheduler does the following: **(i)** makes all subtasks in Q ready, *i.e.*, assigns $H := Q$ (line 17 of PD$^{\mathrm{Q}}$); **(ii)** removes the array of heaps R (line 20); and **(iii)** selects for execution h subtasks, where h is the number of idle processors (lines 22–27). Rule A is implemented by lines 3–15 of *Update* as follows. Let T_i be the subtask of T mentioned in the quick-release rules that has a deadline after $u+1$, where Rule A is applied at u, and let T_k be its successor. If there is no hole in slot t, where T_i is scheduled (which would be before slot u), then T_k is simply inserted into heap Q (line 5 of *Update*), where it remains until it is either scheduled or a slot with a hole is encountered. (Q is ordered according to PD^2 priorities taking into account the offset for each subtask's PF-window; we assume that the *Insert* routine takes this offset as a parameter for insertion into a heap.) If slot t has a hole (in which case $t = u$), then T_k is inserted into Q (lines 9 and 12 of *Update*) as well as into the appropriate release heap $R[u]$ depending on whether T is $(t+1)$- or $(t+2)$-releasable at time t (lines 8, 11, and 16–18 of *Update*). Thus, whenever any subtask T_i is scheduled, its successor subtask T_k is inserted into Q as well as $R[t']$ where $t' = r(T_k)$ (line 5 of *Requeue*; note that $r(T_k)$ may have been unchanged by the quick-release rules, or changed as shown in lines 8 and 11 of *Update*).

```
variables
    H: priority-ordered heap of tasks;
    R: array [0..∞] of priority-ordered heap of tasks;
    Q: priority-ordered heap of tasks;
    J: priority-ordered heap of tasks;
    T: task;
    S: array [1..M] of task;
    t: integer;
    r′: integer;
    r: integer;
    n: integer;
    k: integer

Update(T: task, t: integer)
1:  r := T's current release;
2:  r′ := T's next release;
3:  if T's current deadline is greater than t + 1 then
4:      if t has no hole then /* refer to line 15 in PD^Q */
5:          Insert(Q, T, max(0, r + x(T) − t))
6:      else if T is light then
7:          if T's previous deadline is at most t then
8:              r′ := min(r′, max(t + 1, r + x(T)));
9:              Insert(Q, T, r′ − t)
10:         else if T's previous deadline is at t + 1 then
11:             r′ := min(r′, max(t + 2, r + x(T)));
12:             Insert(Q, T, r′ − t)
13:         fi
14:     fi
15: fi;
16: Update T's next release to be r′;
17: Update T's deadline, b-bit, and group deadline fields;
18: Requeue(T, r′)

Requeue(T: task, t: integer)
1:  if R[t] does not exist then
2:      R[t] := MakeHeap()
3:  fi;
4:  Determine T's priority at time t;
5:  Insert(R[t], T, 0)
```

```
Algorithm PD^Q
1:  H := MakeHeap();
2:  Q := MakeHeap();
3:  R := MakeHeapArray();
4:  t := 0;
5:  when next time slot begins do
6:      H := Merge(H, J);
7:      free(J);
8:      n := 0;
9:      while n < M and H is non-empty do
10:         T := ExtractMin(H);
11:         Delete(Q, T);
12:         Schedule task T in slot t;
13:         S[n] := T;
14:         n := n + 1
15:     od;
16:     if n < M then  /* there is a hole in slot t */
17:         H := Q;
18:         free(Q);
19:         Q := MakeHeap();
20:         free(R);
21:         R := MakeHeapArray();
22:         while n < M and H is non-empty do
23:             T:= ExtractMin(H);
24:             Schedule task T in slot t;
25:             S[n] := T;
26:             n := n + 1
27:         od
28:     fi;
29:     k := n;
30:     while k > 0 do
31:         Update(S[k], t);
32:         k := k − 1
33:     od;
34:     if R[t + 1] is non-empty then
35:         H := Union(H, R[t + 1])
36:     fi;
37:     t := t + 1
38: od
```

Figure 5: Implementation of the PD^{Q} scheduling algorithm.

4 Correctness of PD^{Q}

Before continuing, we introduce some terminology that we use in the rest of the paper.

Terminology. We use the term *AB-GIS task system* to refer to GIS task system with subtask releases that are modified according to Rules A and B. We use the term *AC-GIS task system* to refer to a GIS task system with subtask releases that are modified according to Rule A and Rule C as follows.

Rule C: $e(T_k)$ may be assigned any value in $[t+1, r(T_k)]$.

In other words, in an AC-GIS task system, subtask deadlines are left-shifted when a slot with a hole in encountered, but the idle capacity in that slot is not utilized.

An *instance* of a task system is obtained by specifying a valid release time and eligibility time for each subtask. Note that the deadline of a subtask is automatically determined once its release time is fixed (refer to (2)). We

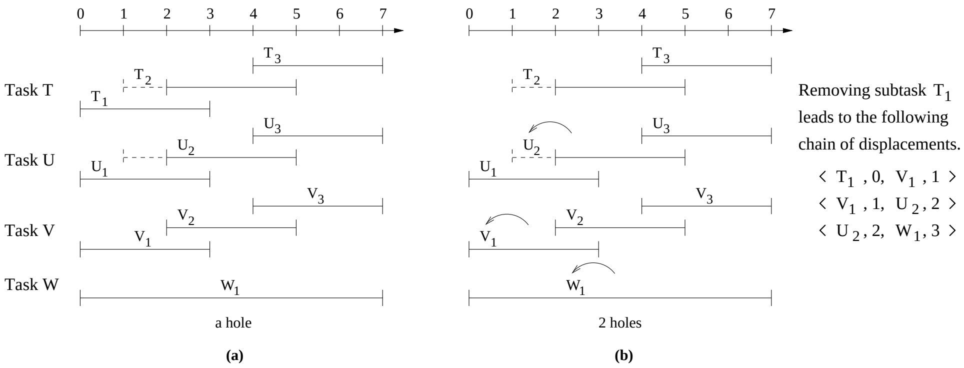


Figure 6: A schedule for three tasks of weight 3/7 and one task of weight 1/7 on two processors. Only subtasks T_2 and U_2 are eligible before their PF-windows. Inset **(b)** illustrates the displacements caused by the removal of subtask T_1 from the schedule shown in inset **(a)**.

say that an algorithm *correctly schedules* a task system instance if it ensures that all subtask deadlines are met.

Proof overview. We prove that PD^{Q} correctly schedules any AB-GIS task system in two steps.

Step 1: If PD^{Q} correctly schedules any AC-GIS task system, then it correctly schedules any AB-GIS task system.

Step 2: PD^{Q} correctly schedules any AC-GIS task system.

In Step 1, we show that applying Rule B instead of Rule C cannot cause a missed deadline. In Step 2, we prove that an assumption to the contrary leads to a contradiction. In particular, we start with an AC-GIS task system τ that misses a deadline under PD^{Q}. We then convert it into a dynamic GIS task system that satisfies (J) and (L) and yet misses a deadline under PD^2. Thus, we obtain a contradiction of Theorem 1.

In transforming one schedule to another, we need to consider task systems obtained by removing subtasks from τ. We now present certain results about subtask removals that are used in our proofs.

Displacements. By definition, the removal of a subtask from one instance of a GIS (or an AC-GIS) task system results in another valid instance. Let $X^{(i)}$ denote a subtask of any task in a GIS task system τ. Let S denote any schedule of τ obtained by an EPDF-based algorithm. Assume that removing $X^{(1)}$ scheduled at slot t_1 in S causes $X^{(2)}$ to shift from slot t_2 to t_1, where $t_1 \neq t_2$, which in turn may cause other shifts. We call this shift a *displacement* and denote it by a four-tuple $\langle X^{(1)}, t_1, X^{(2)}, t_2 \rangle$. A displacement $\langle X^{(1)}, t_1, X^{(2)}, t_2 \rangle$ is *valid* iff $e(X^{(2)}) \leq t_1$. Because there can be a cascade of shifts, we may have a *chain* of displacements, as shown in Fig. 6.

The lemmas below concern displacements and holes, and are proved in [14]. Though these were proved in the context of GIS task systems, they apply to AC-GIS task systems as well. Lemma 1 states that a subtask removal can only cause left-shifts, as in Fig. 6(b). Lemma 2 indicates when a left-shift into a slot with a hole can occur. Here, S denotes a schedule for a task system τ obtained by an EPDF-based algorithm (such as PD^2 and PD^{Q}).

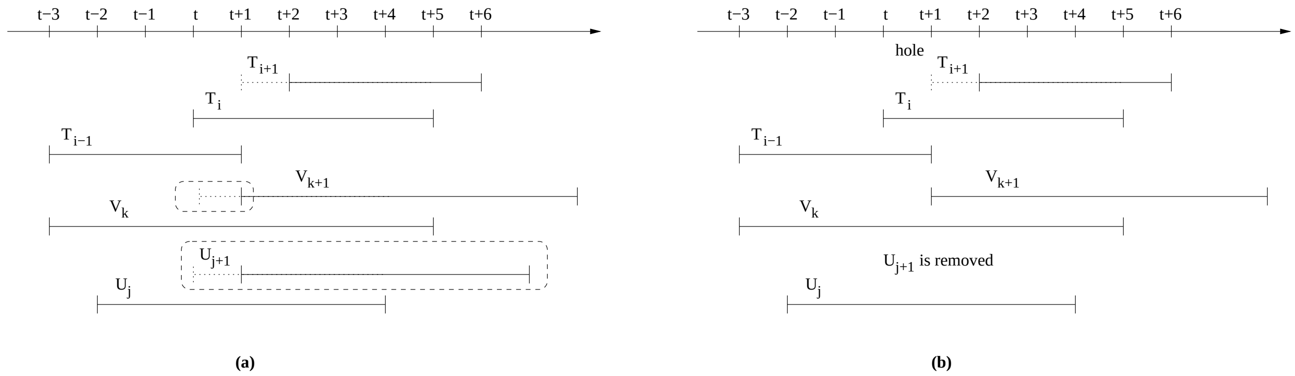


Figure 7: **(a)** There is a hole in slot t and Rules A and B are applied to tasks T, U, and V, as shown. T is $(t+2)$-releasable at t, while U and V are $(t+1)$-releasable. U_{j+1} is scheduled in slot t. **(b)** The schedule resulting after removing U_{j+1} and changing the eligibility time of V_{k+1} to $t+1$. No subtask can shift from $t' \geq t+1$ into slot t. Thus, there is a hole in slot t.

Lemma 1 *Let $X^{(1)}$ be a subtask that is removed from τ, and let the resulting chain of displacements in S be $C = \Delta_1, \Delta_2, \ldots, \Delta_k$, where $\Delta_i = \langle X^{(i)}, t_i,\ X^{(i+1)}, t_{i+1}\rangle$. Then $t_{i+1} > t_i$ for all $i \in \{1, \ldots, k\}$.*

Lemma 2 *Let $\Delta = \langle X^{(1)}, t_1, X^{(2)}, t_2\rangle$ be a valid displacement in S. If $t_1 < t_2$ and there is a hole in slot t_1 in that schedule, then $X^{(2)}$ is $X^{(1)}$'s successor in τ.*

4.1 Step 1

Lemma 3 *If* PD$^{\mathrm{Q}}$ *correctly schedules all feasible AC-GIS task systems, then it correctly schedules all feasible AB-GIS task systems.*

Proof: We transform an AB-GIS task system instance τ to a *corresponding* AC-GIS task system instance τ' by applying two modifications: **(i)** if the PF-window of any subtask is left-shifted by Rule A due to a hole in slot t, and it is scheduled in slot t, then such a subtask is removed; **(ii)** if the PF-window of any subtask is left-shifted by Rule A due to a hole in slot t, and it is not scheduled in slot t, then Rule C is applied to it instead of Rule B. Fig. 7 illustrates these modifications. Since Rules A and C are applied, τ' is an AC-GIS task system. Further, because these modifications affect only the subtasks that are early-released and also scheduled where there was a hole, the rest of the schedule remains the same. Thus, τ would miss a deadline under PD$^{\mathrm{Q}}$ if and only if τ' would. Hence, if PD$^{\mathrm{Q}}$ correctly schedules τ', then it correctly schedules τ as well. □

4.2 Step 2

Lemma 4 PD$^{\mathrm{Q}}$ *correctly schedules any AC-GIS task system.*

Proof: We prove Lemma 4 by contradiction. We start with the assumption that PD$^{\mathrm{Q}}$ misses a deadline for some AC-GIS task system instance τ, and then inductively transform τ to a dynamic GIS task system instance

τ' that satisfies (J) and (L), and misses a deadline under PD^2. During this transformation, we produce several intermediate schedules in which all subtask releases up to a certain time are in accordance with the definition of a GIS task system. To facilitate this, we call a schedule *t-GIS-compliant* if all subtask releases that are at most t satisfy (2) and (3).

Let t_d be the earliest time at which a deadline is missed by any AC-GIS task system instance under PD$^{\mathrm{Q}}$. Let τ be an AC-GIS task system instance that misses a deadline at t_d, and let S be its PD$^{\mathrm{Q}}$ schedule. Without loss of generality, we assume that τ satisfies the following property.

$$\text{for every subtask } T_i \text{ in } \tau,\ e(T_i) = min(r(T_i), t), \tag{4}$$

where t is the time at which T_i is scheduled in S. If $e(T_i) < r(T_i)$, and if T_i is scheduled at $t < r(T_i)$, then this assumption has the effect of redefining $e(T_i)$ to be t. This does not affect the schedule produced by PD$^{\mathrm{Q}}$.

Before continuing with the proof of Lemma 4, we first establish the following.

Lemma 5 *Suppose that S is t-GIS-compliant and there is a hole in slot t. Let U_j be a subtask in τ that is scheduled at or before t such that $d(U_j) > t+1$. Then U_j can be removed from τ without causing the missed deadline at time t_d to be met.*

Proof: Let τ' be the task system instance obtained by removing U_j from τ, and let S' be its PD$^{\mathrm{Q}}$ schedule. Let the chain of displacements caused by removing U_j be $\Delta_1, \Delta_2, \ldots, \Delta_k$, where $\Delta_i = \langle X^{(i)}, t_i,\ X^{(i+1)}, t_{i+1}\rangle$, $X^{(1)} = U_j$, and t_1 is the time at which U_j is scheduled in S. (Note that by the statement of the lemma, $t_1 \le t$.) By Lemma 1, $t_{i+1} > t_i$ for all $i \in [1, k]$. Note that at slot t_i, the priority of subtask $X^{(i)}$ is higher than the priority of $X^{(i+1)}$, because $X^{(i)}$ was chosen over $X^{(i+1)}$ in S. Thus, because $X^{(1)} = U_j$, by the statement of the lemma, we have the following: for each subtask $X^{(i)}, i \in [1, k+1]$, $d(X^{(i)}) > t+1$. Therefore, by (2) and (3), the property (X) stated below follows. This property holds even if the successor subtask $X^{(i,succ)}$ has been quick-released, because S is t-GIS-compliant (hence, all subtask releases up to time t satisfy (2) and (3)).

(X) For all $i \in [1, k+1]$, if $X^{(i,succ)}$ is the successor subtask of $X^{(i)}$, then $r(X^{(i,succ)}) \ge t+1$.

We now show that the displacements do not extend beyond slot t, which implies that a deadline is still missed at t_d in S', as required. Suppose that these displacements do extend beyond slot t, *i.e.*, $t_{k+1} > t$. Let h be the smallest $i \in [2, k+1]$ such that $t_i > t$. Since $t_1 \le t$, we have $t_{h-1} \le t$. Further, because Δ_{h-1} is valid,

$$e(X^{(h)}) \le t_{h-1}. \tag{5}$$

Now, if $t_{h-1} < t$, then by the above expression, $X^{(h)}$ is eligible at t. Because there is a hole in slot t, this implies that $X^{(h)}$ should have been scheduled at t in S instead of later at t_h. Therefore, $t_{h-1} = t$. (Refer to Fig. 8.)

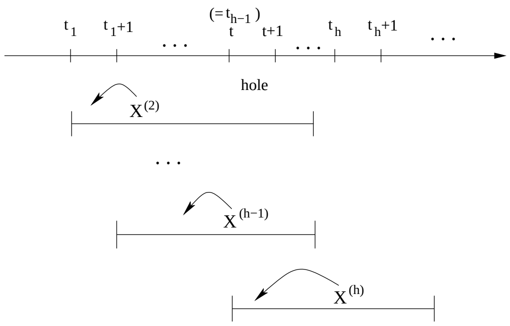


Figure 8: Lemma 5. $X^{(h)}$ must be the successor of $X^{(h-1)}$ because there is a hole in slot t.

Because there is a hole in slot t, by Lemma 2, $X^{(h)}$ is the successor of $X^{(h-1)}$. Thus, by (X), $r(X^{(h)}) \geq t+1$. Because $X^{(h)}$ is scheduled in slot t_h and $t_h > t$, by (4), we have $e(X^{(h)}) \geq t+1 > t_{h-1}$. This contradicts (5). Thus, no subtask scheduled after t can get left-shifted, and hence, a deadline is still missed at t_d in S'. □

Proof of Lemma 4 (continued): We now show by induction that a t_d-GIS-compliant schedule exists in which a deadline is missed at t_d, contradicting Theorem 1.

Base case. Because any schedule is 0-GIS-compliant, S is 0-GIS-compliant.

Induction step. We assume that S is t-GIS-compliant and prove that a $(t+1)$-GIS-compliant schedule exists that also has a missed deadline at time t_d. If all subtask releases at time $t+1$ are in accordance with (2) and (3), then S is $(t+1)$-GIS-compliant. Otherwise, there exists a subtask for which Rule A is applied, and its PF-window has been shifted to begin at $t+1$. Therefore, corresponding to this quick-release, there exists a slot $u \in [0, t+1)$ with a hole, and a task T that is $(u+1)$- or $(u+2)$-releasable at u. (Because we are using Rule C, the hole at u is not eliminated by quick releasing.) Let T_i be the subtask of T mentioned in the quick-release rules that has a deadline after $u+1$, *i.e.*,

$$d(T_i) > u+1. \tag{6}$$

Let T_k be the successor subtask of T_i that has its PF-window shifted to $t+1$. We consider two cases depending on whether T is $(u+2)$- or $(u+1)$-releasable at u. (In the former case, $t+1 \geq u+2$, and in the latter, $t+1 \geq u+1$.)

Case 1: T is $(u+2)$-releasable at u. In this case, by definition, T is light, T_i is scheduled in slot u, and T_i's predecessor has a deadline at time $u+1$. By Lemma 5, T_i can be removed without causing the missed deadline at t_d to be met.

We construct τ' from τ by creating two new tasks U and V with the same weight as T as follows: U leaves

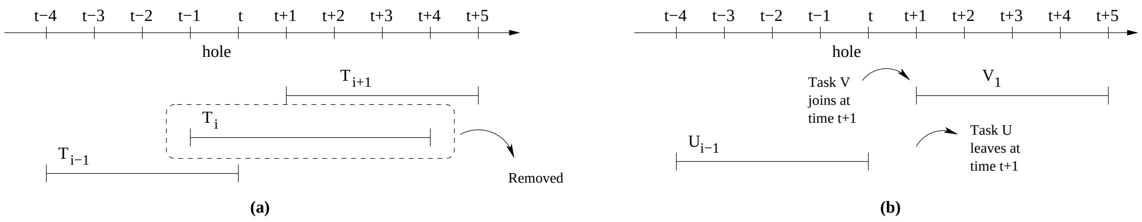


Figure 9: There is a hole in slot $t-1$. **(a)** Case 1 in the proof of Lemma 4. Task T is $(t+1)$-releasable at $t-1$. T_{i-1}, T_i, and T_{i+1} are assumed to be consecutive subtasks of T (no subtasks are missing). **(b)** The new task system obtained by breaking task T into two tasks U and V.

the system at time $u+2$, and V joins the system at $r(T_k)$, which equals $t+1$, by assumption. (Recall that $t+1 \geq u+2$.) Further, U consists of all subtasks of T (if any) up to T_i's predecessor, and V consists of all the subtasks of T after T_i. In other words, τ' is obtained by removing T_i from τ. (Fig. 9 illustrates this for $u = t-1$.)

Since $wt(V) = wt(T)$, V satisfies the join condition (J). Because T is light and T_i's predecessor has a deadline at $u+1$, by (L), T can be allowed to leave at time $u+2$. Thus, U satisfies the leave condition (L). In other words, all subtask releases of U and the first subtask release of V (trivially) are in accordance with (2) and (3).

Case 2: T is $(u+1)$-releasable at u. In this case, either T_i is scheduled before u or it is scheduled in slot u, and its predecessor subtask T_j has a deadline at or before u. Therefore, by Lemma 5, T_i can be removed without causing the missed deadline at t_d to be met.

As in Case 1, we construct a new task system instance τ' from τ by creating two new tasks U and V with the same weight as T as follows: U leaves the system at time $u+1$, and V joins the system at $r(T_k)$, which again equals $t+1$. (Recall that $t+1 \geq u+1$.) U consists of all the subtasks of T (if any) up to T_i's predecessor, and V consists of all the subtasks of T after T_i. As in Case 1, V satisfies the join condition (J); we now show that U satisfies the leave condition (L).

Claim 1 *Task U satisfies the leave condition* (L).

Proof of Claim: Because T is $(u+1)$-releasable at time u, there are two possibilities: either **(i)** T_i is scheduled before slot u, or **(ii)** T_i is scheduled in slot u, and its predecessor has a deadline at or before u. If (i) holds, then $r(T_i) < u \leq t$, which, by (2), (3), and the fact that S is t-GIS compliant, implies that $d(T_j) \leq u$ for all $j < i$.

Thus, under both possibilities, the last-scheduled subtask of task U before time u has a deadline at or before u. If U is light, then by (L), U can be allowed to leave at time $u+1$.

Because $r(T_i) < u$ and $d(T_i) > u+1$ (by (6)), the length of T_i's PF-window is at least three. Therefore, if T is heavy and T_j is T_i's predecessor in τ (if its predecessor exists), then $D(T_j) \leq u+1$. Hence, by (L), task T can be allowed to leave at time $u+1$. Thus, U satisfies (L). This concludes the proof of Claim 1. □

By Claim 1, T can be broken up into two tasks U and V that satisfy (L) and (J), respectively, and this modification ensures that all subtask releases of U and V until time $t+1$ satisfy (2) and (3). Further, a deadline is still missed at time t_d. This concludes Case 2.

Repeating this argument for all subtasks that have their PF-windows shifted (by Rule A) to begin at $t+1$, we can obtain a schedule that is $(t+1)$-GIS-compliant. Therefore, by induction, there exists a t_d-GIS-compliant schedule that misses a deadline at t_d. This contradicts Theorem 1. Hence, τ and t_d cannot exist. This completes the proof of Lemma 4. □

By Lemmas 3 and 4, we have the following theorem.

Theorem 2 PD$^{\mathrm{Q}}$ *correctly schedules any AB-GIS task system.*

Consider a task that is always eligible to execute over an interval $[0, t)$, *i.e.*, it is *backlogged* throughout that interval. Note that the Pfair deadline of a subtask T_i is determined based on $wt(T)$. Under QRfair scheduling, Rules A and B only cause these deadlines to be shifted left. Because, under Pfair scheduling, T is guaranteed a share of $\lfloor wt(T) \times t \rfloor$ over $[0, t)$, PD$^{\mathrm{Q}}$ provides the same guarantee. Thus, by Theorem 2, we have the following.

Theorem 3 *If a task T is backlogged throughout the interval $[0, t)$, then* PD$^{\mathrm{Q}}$ *guarantees T a share of at least $\lfloor wt(T) \times t \rfloor$ over the interval $[0, t)$.*

5 Enforcing Maximum Weights

In the previous section, we showed that if there is a hole at slot t, then each task's next subtask window could be left-shifted to begin as early as time $t+1$ or $t+2$ (if the window would otherwise begin later). However, if every subtask of a task is always left-shifted as aggressively as possible, then such a task may be allocated more processing capacity than it can use. Such overallocations can be avoided by specifying a *maximum weight* for each task, which indicates the maximum amount of processing capacity that it can use. The weight $wt(T)$ considered earlier can then been seen as specifying a *minimum weight* for task T.

Fortunately, incorporating maximum weights within PD$^{\mathrm{Q}}$ is not difficult. Rule A given earlier requires that a minimum separation between subtask releases of the same task be respected. To ensure that a task is not over-allocated, we simply have to control left-shifts in a manner that ensures that the separation between consecutive subtask deadlines of that task is in keeping with its maximum weight. It can be shown that a task of a given weight has windows of at most two different lengths. For example, the task in Fig. 3 has windows of length two and three. Thus, there are at most two different window lengths associated with a given maximum weight. The minimum separation between a task's subtask deadlines can therefore be defined by using a constant separation as defined by either the **(i)** smaller or **(ii)** larger window length associated with its maximum weight (if there is only one such length, then these two rules converge); or **(iii)** by using the actual sequence of windows defined by its maximum weight. Using Method (i), a task's actual allocation may be slightly more than that defined by its maximum weight. Using Method (ii), it may be slightly less. Method (iii) will result in a more accurate allocation, but it requires more computational overhead. To handle boundary conditions when implementing Methods (ii) and (iii), a subtask's separation parameter must actually be defined as the minimum of the separation defined by its minimum weight and that described above (otherwise, the task's allocation rate might occasionally dip below that defined by its minimum weight). If maximum weights should never be exceeded, then Method (ii) must be used. For this method, we have the following counterpart of Theorem 3. (Due to certain pathological cases, which for lack of space we do not describe, this theorem may not always hold for Methods (i) and (iii).)

Theorem 4 *If maximum weights are enforced using Method* (*ii*) *and Rule A* (*i.e., subtasks are not early released*), *and if a task T is backlogged throughout the interval* $[0, t)$, *then* PD^{Q} *guarantees T a share of at most* $\lceil maxwt(T) \times t \rceil$ *over the interval* $[0, t)$, *where $maxwt(T)$ is the maximum weight of T.*

Defining useful maximum weights. If the sum of all maximum weights exceeds M (the number of processors), then the capacity allocated to some tasks may be very near their maximum weights, while others may be much less (although the capacity allocated to each task will be at least that defined by its minimum weight). Such a situation is not in keeping with the principle of fairness. The likelihood of unfair allocations can be reduced by specifying an *effective maximum weight* (or *effective max*, for short) for each task so that the sum of all such weights is less than M. The effective max of a task gives the maximum rate at which it can execute, given the current load of the system. Such an effective max can be calculated using the formula

$$\textit{effective max of task } T = wt(T) + \frac{M - W}{X - W} \cdot (maxwt(T) - wt(T)), \tag{7}$$

where W is the sum of all minimum task weights, X is the sum of all *actual* (not effective) maximum task weights, $maxwt(T)$ is the actual maximum weight for task T, and $wt(T)$ is the minimum weight for task T. If $X \geq M$, then $M - W \leq X - W$, and hence, this definition ensures that no task will exceed its maximum weight.[3] Further, $\sum_T \left(wt(T) + \frac{M-W}{X-W} \cdot (maxwt(T) - wt(T)) \right) = W + (M - W) \cdot \frac{X-W}{X-W}$, which simplifies to M. Thus, the sum of all effective maxes is exactly M. It is also clear from the formula that a task's effective max is at least its minimum weight. (This is why the simpler formula $(M/X) \cdot maxwt(T)$ may not always work.) Although other methods could be used for calculating effective maxes, this is the one used in the experiments presented

[3]If effective maxes are used when $X < M$, then a task's effective max will be at least its actual maximum weight. This is really not a problem, however, because the system is underutilized anyway.

in the next section. The results presented there indicate that fair allocations result when using effective maxes defined in this way. An additional advantage is that effective maxes so defined can be efficiently maintained. In particular, the right-hand-side of Equation (7) can be re-written as $\frac{X-M}{X-W} \cdot wt(T) + \frac{M-W}{X-W} \cdot maxwt(T)$. Since the coefficients of $wt(T)$ and $maxwt(T)$ are the same for all tasks, they can be updated globally in $O(1)$ time as the workload changes. With these coefficients, a task's effective max can be computed in $O(1)$ time.

A different definition of an effective max has been given previously by Chandra *et al.* [8]. In their definition, all actual maximum weights are assumed to be unity. This is obviously quite limiting because tasks can easily exist in practice that cannot make use of the entire capacity of a single processor. In work on elastic scheduling [7], non-unity maximum weights were considered, but this work (as presented) pertains only to uniprocessors.

6 Experiments

To empirically evaluate PD^{Q}, we conducted a number of simulations. Each simulated system consisted of eight processors, and 100 randomly-weighted light tasks, each with a light minimum weight and a possibly-heavy maximum weight. Each system was simulated for 30,000 time steps. In enforcing maximum weights, Method (iii) in Sec. 5 was used. Effective maxes were used as maximum weights and determined using Equation (7). One way of evaluating PD^{Q} is by comparing its allocations to that of an ideal scheduler. In each time slot, such an ideal scheduler schedules each eligible task for a fraction of a quantum that is equal to its effective max.

The first experiment was conducted to determine the fraction of its ideal allocation that each task receives under PD^{Q}. We refer to this fractional value as an *allocation ratio*. Fig. 10(a) shows the allocation ratios for one randomly-generated task set. In this figure, such a ratio is plotted for each of the 100 tasks in the task set. A ratio of one means that the corresponding task received exactly the amount of processing time that it should have received according to the ideal scheduler (over all 30,000 time steps). As can be seen, tasks ranged from being underallocated by 3% to being overallocated by 4% (relative to its effective max).

To confirm that these results were typical, we randomly generated three hundred different task sets and computed the average value by which a task is underallocated. Specifically, this value was computed for each task set by averaging the allocation ratios of all underallocated tasks in that task set. The results are shown in Fig. 10(b). In this graph, the x-axis ranges over all 300 task sets. As this graph shows, no system had an average underallocation of more than 3.5%. Furthermore, in most systems, this average was between 1% and 2%.

To determine the effectiveness of PD^{Q} in a dynamic system, we simulated a number of systems in which tasks leave and join dynamically. Insets (c) and (d) of Fig. 10 depict results pertaining to one such system. In this system, there were 97 dynamic tasks and three initially-present static tasks. Each dynamic task joined the

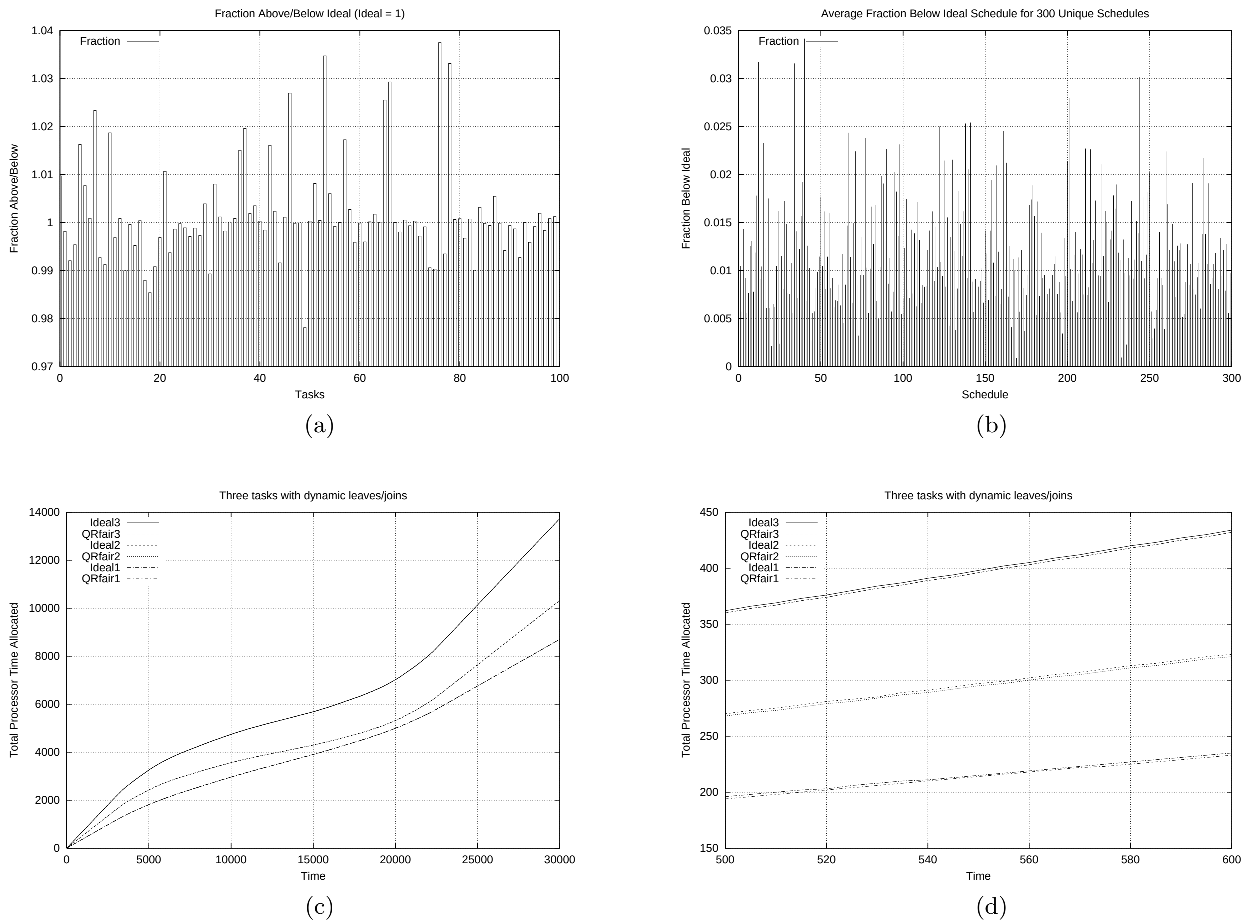
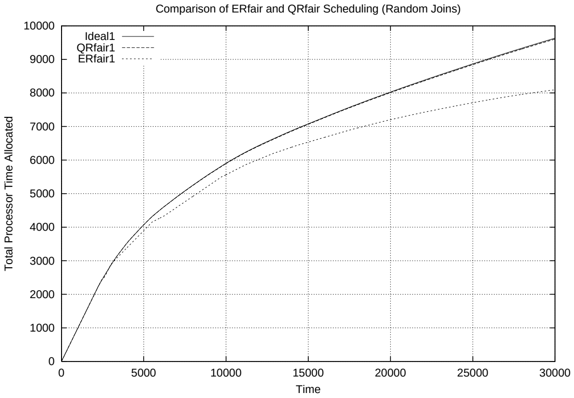


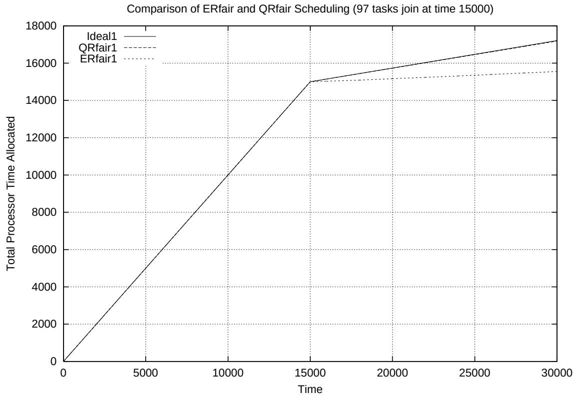
Figure 10: Simulation results.

system at some randomly-selected time, and then left at a later randomly-selected time. The graph in Fig. 10(c) depicts the allocations of the three static tasks as a function of time. The graph actually depicts both the $\mathrm{PD^Q}$ and ideal allocation for each of these tasks. However, these allocations almost coincide for each task, so it is difficult to see that six curves have actually been plotted. The graph in Fig. 10(d) depicts a magnified view of the graph in Fig. 10(c) over the time interval $[500, 600]$. Even in this view, the difference between the $\mathrm{PD^Q}$ and ideal allocations is very small. This experiment was repeated a number of times with other task sets, and similar results were obtained each time. Unfortunately, we cannot present these other results due to space limitations.

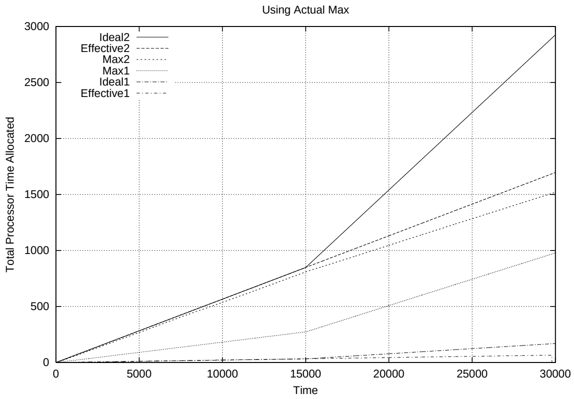
The next experiments presented here were conducted to compare $\mathrm{PD^Q}$ to ER scheduling, which penalizes tasks for having used spare capacity in the past. The results from two such experiments are shown in the two graphs in Fig. 11(a)-(b). As before, each system consisted of three initially-present static tasks, and 97 dynamic tasks. In the experiment depicted in Fig. 11(a), each dynamic task joined at some randomly-selected time (and did not leave). In that depicted in Fig. 11(b), each such task joined at time 15,000 (and did not leave). To avoid clutter, we have only depicted in each graph the allocations for one of the static tasks. (The curves for



(a)



(b)



(c)

Figure 11: Simulation results (continued).

the other tasks show similar trends.) Three curves have been plotted in each graph, showing the depicted task's PD^{Q} allocation, its ER allocation, and its ideal allocation. As before, the ideal allocation in each case so closely coincides with the PD^{Q} allocation that it is hard to discern that there are actually three curves in each graph. It can be seen that the use of only ER scheduling to utilize spare capacity results in underallocation. This is due to the problems with ER scheduling noted before. As before, a number of other experiments were conducted showing similar results, but these other results have been omitted due to space constraints.

Our final experiment illustrates the importance of using effective maxes instead of actual maximum weights if the latter results in an overutilized system and fairness is a concern. The graph in Fig. 11(c) depicts the allocations of two tasks as a function of time. The graph shows the allocations of each task under an ideal scheduler and under PD^{Q}, using actual maximum weights and effective maxes. At time 15,000, half of the tasks drop their rates in half. This information is not conveyed to the PD^{Q} scheduler (that is, this change is not registered by having these tasks leave with their old weights and then re-join with new weights); however, we assume that it *is* conveyed to the ideal scheduler. Two behaviors can be observed. First, PD^Q does not respond to the change when effective maxes are used. Second, even though PD^Q is responsive when actual maximum weights are used, it is significantly *unfair*: task 1 receives substantially more processor time then in the ideal schedule, while task 2 receives substantially less. Note that if such a rate change were registered with the scheduler, then new effective maxes could be computed, and PD^Q would then more accurately track the ideal scheduler. As before, a number of experiments showing similar results were performed that are not reported here.

In all experiments that we conducted in which effective maxes were used and dynamic changes were registered with the scheduler, PD^Q and ideal allocations were virtually indistinguishable. While more work is needed to thoroughly evaluate PD^Q, this evidence suggests that PD^Q performs remarkably well in allocating spare capacity.

7 Conclusions

In this paper, we have introduced a new notion of multiprocessor fairness, called quick-release fair scheduling, in which spare processing capacity is allocated without penalizing tasks for having used spare capacity in the past. We have also presented a quick-release scheduling algorithm, PD^Q, and have formally proved it correct. This algorithm has the same asymptotic time complexity as the most efficient Pfair scheduling algorithm currently known. To the best of our knowledge, PD^Q is the first fair multiprocessor scheduling algorithm, with provable properties concerning the allocation decisions it makes, that supports both minimum and maximum task weights. Moreover, unlike prior work, we have not required all maximum weights to be unity.

The results of this paper give rise to several questions that warrant further investigation. For example, it would be interesting to consider other ways of defining maximum weights, as well as efficient mechanisms for maintaining such weights as the workload changes. In addition, the performance results presented in the previous section suggest that the allocations made by PD^Q when using our definition of an effective max very closely track those that would be made by an ideal scheduler. This gives rise to the possibility that PD^Q may have *provable* lag bounds. While such bounds may be quite difficult to obtain, they certainly warrant further research.

References

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

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

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

[5] 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*, 64(1):43–51, October 1997.

[6] J. Bennett and H. Zhang. WF^2Q: Worst-case fair queueing. In *Proceedings of IEEE INFOCOM'96*, pages 120–128, March 1996.

[7] G. Buttazzo, G. Lipari, and L. Abeni. Elastic task model for adaptive rate control. In *Proceedings of the 19th IEEE Real-time Systems Symposium*, pages 286–295, December 1998.

[8] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI 2000)*, pages 45–58, October 2000.

[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 IEEE Real-time Technology and Applications Symposium*, pages 3–14, June 2001.

[10] S.J. Golestani. A self-clocked fair queueing scheme for broadband applications. In *Proceedings of IEEE INFOCOM '94*, pages 636–646, April 1994.

[11] P. Goyal, H. Vin, and H. Cheng. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. In *Proceedings of ACM SIGCOMM'96*, pages 157–168, August 1996.

[12] A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical Report MIT/LCS/TR-297, Massachusetts Institute of Technology, 1983.

[13] A. K. Parekh. *A Generalized Processor Sharing Approach To Flow Control in Integrated Services Networks*. PhD thesis, MIT, 1992.

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

[15] 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.

[16] 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, 1996.