Fast Scheduling of Periodic Tasks on Multiple Resources

Sanjoy K. Baruah [1] *Johannes E. Gehrke* [2] *C. Greg Plaxton* [2]

Abstract

Given n periodic tasks, each characterized by an execution requirement and a period, and m identical copies of a resource, the *periodic scheduling problem* is concerned with generating a schedule for the n tasks on the m resources. We present an algorithm that schedules every feasible instance of the periodic scheduling problem, and runs in $O(\min\{m \lg n, n\})$ time per slot scheduled.

1 Introduction

Given a set Γ of n tasks, where each task x is characterized by two integer parameters $x.e$ and $x.p$, and m identical copies of a resource, a *periodic schedule* is one that allocates a resource to each task x in Γ for exactly $x.e$ time units in each interval $[k \cdot x.p, (k+1) \cdot x.p)$ for all k in $\mathbf{N}$, subject to the following constraints:

> *Constraint 1:* A resource can only be allocated to a task for an entire "slot" of time, where for each i in $\mathbf{N}$ slot i is the unit interval from time i to time $i+1$.
>
> *Constraint 2:* No task may be allocated more than one copy of the resource in any single slot.

The problem of constructing periodic schedules for such task systems was discussed by Liu in 1969 [6], and is called the *(multiple resource) periodic scheduling problem.* It has been shown [3, 5] that an instance of the periodic scheduling problem is *feasible* (i.e., has a periodic schedule) if and only if $(\sum_{x \in \Gamma} x.e/x.p) \leq m$.

Given a feasible instance of the periodic scheduling problem, a *schedule generation algorithm* performs a (possibly empty) *pre-processing phase* followed by an infinite *execution phase.* No output is produced during the pre-processing phase. During the execution phase, the algorithm produces an infinite sequence of outputs $\langle X_i : i \geq 0 \rangle$, where X_i is the subset of up to m tasks scheduled (i.e., assigned one copy of the resource) in slot i. (Note that any output-free prefix of the computation may be designated as the preprocessing phase.) Let t_i denote the elapsed running time (in the usual RAM model) between the beginning of the execution phase and the time at which output X_i is produced, $i \geq 0$. Also, let $t_{-1} = 0$. Then for any schedule generation algorithm $\mathcal{A}$, we define the *per-slot time complexity* of $\mathcal{A}$ as the maximum over all feasible instances and over all $i \geq 0$ of $t_i - t_{i-1}$. We further define the *pre-processing time complexity* of $\mathcal{A}$ as the maximum running time of the pre-processing phase over all feasible instances. A schedule generation algorithm is *polynomial time* if and only if both the per-slot time complexity and the pre-processing time complexity are polynomial in the input size.

Because all of the scheduling algorithms considered in this paper are schedule generation algorithms, for the sake of brevity we hereafter use the term "scheduling algorithm" to mean "schedule generation algorithm". Furthermore, because all of the scheduling algorithms that we consider have $O(n)$ pre-processing time complexity, we will focus our attention on minimizing slot time complexity. Accordingly, throughout the paper, every time bound given for a scheduling algorithm should be assumed to be a bound on the slot time complexity unless otherwise specified.

A simple yet powerful idea for solving many scheduling problems involving deadlines is to give priority to the task with the earliest associated deadline, breaking ties arbitrarily. For example, the Earliest Deadline algorithm of Liu and Layland [7] schedules any feasible instance of the *single resource* (i.e., $m = 1$) periodic scheduling problem. The same algorithm can also be used to solve a relaxed version of the (multiple resource) periodic scheduling problem in which Constraint 2 is eliminated. However, the Earliest Deadline algorithm does not solve the periodic scheduling problem (i.e., it does not schedule every feasible instance). Informally, the difficulty is that neglecting a particular task x with a "later" deadline can eventually lead to a state in which the deadline of x cannot be met with-

[1] Department of Electrical Engineering and Computer Science, The University of Vermont, Burlington, VT 05405. Supported in part by NSF grants OSR-9350540 and CCR-9410752, and UVM grant PSCI94-3.

[2] Department of Computer Science, University of Texas at Austin, Austin, TX 78712. Supported by the Texas Advanced Research Program under Grant No. 003658–461.

out violating Constraint 2. One might expect that it would be relatively straightforward to "patch up" the Earliest Deadline algorithm to take care of such difficulties. However, this turns out to be a non-trivial problem. In fact, prior to the recent work of Baruah et al. [1], no polynomial time algorithm of any kind was known for the periodic scheduling problem.

In [1], an algorithm is presented for the periodic scheduling problem with a running time that is linear in the size of the input in bits, that is,

$$O\left(\sum_{x\in\Gamma} \lceil \lg(x.p+1) \rceil\right). \tag{1}$$

(Note that the preceding sum is at least n, since there are n tasks and the period of each task is at least 1.) This algorithm is based on a measure of fairness in resource allocation called "proportionate progress". Given an instance of the periodic scheduling problem, define the *weight* of any task x in Γ as $x.w \stackrel{\text{def}}{=} x.e/x.p$. Informally, a schedule is said to maintain proportionate progress if and only if each task is scheduled resources in proportion to its weight. Formally, at every time t in $\mathbf{N}$, each task x must have been scheduled either $\lfloor x.w \cdot t \rfloor$ or $\lceil x.w \cdot t \rceil$ times. This requirement is called *proportionate fairness* or *P-fairness.* P-fairness is a strictly stronger condition than periodic scheduling, in that any P-fair schedule is periodic while the converse does not hold. It has been shown, however, that every feasible instance of the periodic scheduling problem has a P-fair schedule [1].

Although Algorithm PF and the Earliest Deadline algorithm bear little obvious resemblance to one another (e.g., they tend to produce radically different schedules even in the case $m = 1$), there is in fact an interesting relationship between these two algorithms. In particular, as we now explain, Algorithm PF may be viewed as a generalization of Earliest Deadline. Under Earliest Deadline, the deadline associated with each allocation to a given task during a particular period is the same, namely, the last slot in the period. Under Algorithm PF, additional "quasi-deadlines" are introduced, and each allocation of a given task has a distinct associated quasi-deadline. Furthermore, these quasi-deadlines are roughly evenly-spaced, where the precise spacing is determined by P-fairness considerations. Algorithm PF then behaves as an "earliest quasi-deadline" algorithm, with the following important caveats:

- There are two kinds of quasi-deadlines that can occur, which may be thought of as "sharp" and "fuzzy". If the ith allocation of a task x has a sharp (resp., fuzzy) quasi-deadline at slot t, then the $(i+1)$th allocation to task x is not allowed to occur prior to slot $t+1$ (resp., t). Thus, the slot at which a given allocation can occur is bounded both from above and below. It turns out that the sharp deadlines are very easy to deal with, informally because a sharp quasi-deadline decouples the scheduling of the two adjacent allocations.

- If the earliest quasi-deadlines associated with two tasks x and y are both fuzzy and occur in the same time slot, we cannot simply break the tie arbitrarily in order to assign priority to either x or y. (Feasible instances of the periodic scheduling problem are known for which this heuristic fails.) Algorithm PF breaks such a tie by comparing the second earliest quasi-deadlines of the two tasks. If there is still a tie, the third earliest quasi-deadlines are compared, and so on. (If all future quasi-deadlines are the same for both tasks, then the tie can be broken arbitrarily.) Algorithm PF uses a GCD-like procedure to efficiently implement the tie-breaking procedure described above. (A naive implementation that directly compares successive quasi-deadlines until the tie is broken would not yield a polynomial-time algorithm for the periodic scheduling problem.)

While the GCD-like tie-breaking procedure alluded to above is quite fast, it does not run in constant time. Instead, the running time of this procedure is linear in the number of bits in the binary representation of the relevant task periods. As a result, the running time of Algorithm PF (see Equation (1)) can exceed $\Theta(n)$ by an arbitrarily high multiplicative factor.

In this paper, we describe and prove correct Algorithm PD, an algorithm for the periodic scheduling problem with running time $O(\min\{m \lg n, n\})$. (The letters "PD" stand for "pseudo-deadline", a term very closely related to the "quasi-deadline" notion discussed above, and that is formally defined in Section 4.) The most efficient algorithms previously known for solving the uniprocessor periodic scheduling problem (e.g., efficient implementations of the Earliest Deadline algorithm) run in $O(\lg n)$ time; note that Algorithm PD matches this time bound for uniprocessor scheduling. Like Algorithm PF, Algorithm PD solves the periodic scheduling problem by generating a P-fair schedule for every feasible instance. (To the best of our knowledge, no previously known algorithm generates P-fair schedules even in the case of a single resource.)

Our work builds on that of [1] by making use of the correctness of Algorithm PF to exhibit a constant-time tie-breaking procedure that is sufficient to maintain P-fairness. In essence, our approach is to limit the tie-breaking procedure to look only a constant number of pseudo-deadlines into the future. Informally, the tie-breaking procedure of Algorithm PF has a finer "resolution" than that of Algorithm PD and, as a result, the two algorithms do not always make the same scheduling decisions. In spite of this, we are able to argue that the schedules generated by Algorithms PD and PF are closely related.

Our main technical contribution is the proof of Theorem 1, which establishes that, like Algorithm PF, Algorithm PD produces a P-fair schedule for any feasible instance of the periodic scheduling problem. (The running time analysis of Algorithm PD is, by contrast, entirely straightforward.) The following new concepts have served to motivate both the definition of Algorithm PD as well as its proof of correctness:

- We take advantage of the duality that exists between "heavy" tasks (those with weight greater than or equal to 1/2) and "light" tasks (those with weight less than or equal to 1/2). Note that a heavy (resp., light) task is scheduled (resp., not scheduled) more often than it is not scheduled (resp., scheduled). As a consequence, when dealing with heavy (resp., light) tasks, it seems to be more important to consider the next time slot by which the task must not be scheduled (resp., be scheduled) than to consider the next time slot by which the task must be scheduled (resp., not be scheduled). While Algorithm PF associates a quasi-deadline with each allocation of every task, Algorithm PD associates a pseudo-deadline with each allocation (resp., "non-allocation") of every light (resp., heavy) task.

- Within the heavy (resp., light) tasks, we further categorize tasks into distinct weight classes. More specifically, all heavy (resp., light) tasks x sharing a common value of $\lfloor 1/(1-x.w) \rfloor$ (resp., $\lfloor 1/x.w \rfloor$) form a single weight class. Note that tasks belonging to the same weight class are scheduled with approximately the same frequency.

- We analyze the state of a schedule generation algorithm in terms of a two-dimensional tableau of integer counts that specifies, for each weight class and for each future slot, the number of associated pseudo-deadlines that have been satisfied to this point in the schedule. We prove that such a tableau provides enough information to maintain P-fairness, even though it does not encode the exact weight of any particular task.

- We precisely characterize the manner and degree to which the state of Algorithm PD can deviate from that of Algorithm PF, given that both algorithms have scheduled exactly the same number of slots. This characterization is expressed in terms of the tableau of count information mentioned above. We find that the deviation is sufficiently small to guarantee that Algorithm PD maintains P-fairness as long as Algorithm PF does. The correctness of Algorithm PD then follows from the fact that Algorithm PF is known to maintain P-fairness [1].

The remainder of this paper is organized as follows. In Section 2, we review the basic definitions associated with P-fairness. In Section 3, we review Algorithm PF of [1]. In Section 4, we present our new algorithm for the periodic scheduling problem, Algorithm PD, along with an outline of the proof of correctness. The full proof of correctness appears in [2]. A straightforward implementation of Algorithm PD has a running time of $O(n)$, which already represents a significant improvement over the performance of Algorithm PF. In Appendix A, we outline a binomial-heap-based implementation of Algorithm PD with a running time of $O(m \lg n)$. Thus, Algorithm PD can be used to solve any instance of the periodic scheduling problem in $O(\min\{m \lg n, n\})$ time. There remain, however, several other important multiple resource scheduling problems for which no efficient solutions are known; in Section 5, we conclude with a general plan for attacking such problems.

2 P-Fairness

In this section we review some concepts introduced in [1]. We start with some conventions:

- Scheduling decisions occur at integral values of *time*, numbered from 0. The unit interval between time t and time $t+1$ will be referred to as slot t, $t \in \mathbf{N}$.

- For integers a and b, let $[a,b) = \{a,\ldots,b-1\}$. Furthermore, let $[a,b] = [a,b+1)$, $(a,b] = [a+1,b+1)$, $(a,b) = [a+1,b)$, and $[a] = [0,a)$.

- We use the variables m and n to denote the number of resources and tasks, respectively, in a given instance of the periodic scheduling problem. Specific tasks are denoted by identifiers x and y, which range over Γ, the set of all tasks.

- Each task x has an integer *period* $x.p$, $x.p > 1$, an integer *execution requirement* $x.e$, $x.e \in (0, x.p)$, and a rational *weight* $x.w = x.e/x.p$. Note that $0 < x.w < 1$. Without loss of generality we assume that $\sum_{x \in \Gamma} x.w = m$.
- Let σ_i denote the ith symbol of string σ, $i \in \mathbf{N}$.

Now some definitions:

- A *schedule* S for an instance of the periodic scheduling problem is a function from $\Gamma \times \mathbf{N}$ to $\{0, 1\}$. We require that $\sum_{x \in [0,n)} S(x,t) \leq m$, $t \in \mathbf{N}$. (In view of our assumption that $\sum_{x \in \Gamma} x.w = m$, we in fact require that $\sum_{x \in [0,n)} S(x,t) = m$.) Informally, $S(x,t) = 1$ if and only if task x is scheduled in slot t.
- A schedule S is defined to be *periodic* if

$$\forall i, x : i \in \mathbf{N}, x \in \Gamma : \textstyle\sum_{t \in [i \cdot x.p]} S(x,t) = i \cdot x.e.$$

- The *lag* of a task x at time t with respect to schedule S, denoted $\mathsf{lag}(S, x, t)$, is defined as:

$$\mathsf{lag}(S, x, t) = x.w \cdot t - \textstyle\sum_{i \in [t]} S(x, i).$$

- A schedule S is defined to be *P-fair* if

$$\forall x, t : x \in \Gamma, t \in \mathbf{N} : -1 < \mathsf{lag}(S, x, t) < 1.$$

- A schedule S is defined to be *P-fair at time t* if there exists a P-fair schedule S' such that

$$\forall x : x \in \Gamma : \mathsf{lag}(S, x, t) = \mathsf{lag}(S', x, t).$$

Every instance of the periodic scheduling problem has a P-fair schedule [1, Theorem 1].

3 A P-Fair Scheduling Algorithm

We now review Algorithm PF, the algorithm defined in [1] that produces a P-fair schedule for any feasible instance of the periodic scheduling problem. We start with some definitions:

- The *characteristic string* of task x, denoted $\alpha(x)$, is an infinite string over $\{-, 0, +\}$ with

$$\alpha_t(x) = \mathrm{sign}(x.w \cdot (t+1) - \lfloor x.w \cdot t \rfloor - 1), \quad t \in \mathbf{N}.$$

- The *characteristic substring* of task x at time t is the finite string

$$\alpha(x, t) \stackrel{\text{def}}{=} \alpha_{t+1}(x)\alpha_{t+2}(x) \cdots \alpha_{t'}(x),$$

where $t' = (\min i : i > t : \alpha_i(x) = 0)$.

- With respect to P-fair schedule S at time t, we say that: task x is *ahead* if and only if $\mathsf{lag}(S, x, t) < 0$; task x is *behind* if and only if $\mathsf{lag}(S, x, t) > 0$; task x is *punctual* if and only if it is neither ahead nor behind.
- With respect to P-fair schedule S at time t, we say that: task x is *tnegru* if and only if x is ahead and $\alpha_t(x) \neq +$; task x is *urgent* if and only if x is behind and $\alpha_t(x) \neq -$; task x is *contending* if and only if it is neither tnegru nor urgent.

Algorithm PF determines which m-subset of the n tasks to schedule in each slot t. As argued in [1], every urgent (resp., tnegru) task must (resp., must not) be scheduled in the current slot in order to preserve P-fairness.

We can define a total order $\succeq$ on the set of contending tasks as follows: $x \succeq y$ if and only if $\alpha(x,t) \geq \alpha(y,t)$, where the comparison between characteristic substrings $\alpha(x,t)$ and $\alpha(y,t)$ is resolved lexicographically with $+ > 0 > -$. (Ties may be broken arbitrarily.)

The behavior of Algorithm PF at each slot t may be summarized as follows:

1. Schedule all urgent tasks.
2. Allocate the remaining resources to the highest-priority contending tasks according to the total order $\succeq$.

Let S_{PF} denote the schedule produced by Algorithm PF on a given instance of the periodic scheduling problem. Schedule S_{PF} is known to be P-fair [1, Theorem 2]. An implementation of the characteristic substring comparison function required by Algorithm PF was presented in [1] and proven to execute in time linear in the size of the binary representation of $\min\{x.p, y.p\}$. This comparison function can be used as the basis for an implementation of Algorithm PF that has a running time as given by Equation (1).

4 Algorithm PD

We now present an algorithm for the periodic scheduling problem that is similar to Algorithm PF, but has a significantly lower running time. Given two tasks x and y at time t, determining which task has higher priority according to Algorithm PF (i.e., whether $x \succeq y$ or $y \succeq x$) takes time linear in the size of the binary representation of $\min\{x.p, y.p\}$. The new algorithm, Algorithm PD, will have as its basis a comparison subroutine that determines the relative priorities of two tasks in constant time.

First, some definitions. The set of tasks is partitioned into light tasks and heavy tasks. A task x is *heavy* if $x.w > 1/2$, and is *light* if $x.w < 1/2$. Each task x with $x.w = 1/2$ may be considered either heavy or light, but not both.

For each task x, we define a string $\beta(x)$ that is closely related to $\alpha(x)$. For light tasks x, $\beta(x) = \alpha(x)$. For heavy tasks x, $\beta(x) = \alpha(x')$ where x' is a task with $x'.w = 1 - x.w$. The intuition behind the string $\beta(x)$ for a heavy task x is as follows: We can obtain a schedule for x from a schedule for x' by allocating a resource to x in exactly those slots where x' is *not* allocated.

A task x is defined to have a *pseudo-deadline at slot* t if $\beta_t(x) = 0$ or $\beta_t(x) = +$. For a light task x, each pseudo-deadline corresponds to a quasi-deadline, i.e., the latest slot by which x must have been allocated the resource a certain number of times in order to maintain P-fairness. For a heavy task x, each pseudo-deadline corresponds to a quasi-deadline of a task x' with $x'.w = 1 - x.w$, i.e., the latest slot by which x must have been *denied* the resource a certain number of times. Let $\delta(x, t)$ denote the least $i > t$ such that task x has a pseudo-deadline at slot i.

Each task x has an integer field $x.k$ determined from $x.e$ and $x.p$ as follows. If x is a light task, then set $x.k = \lfloor x.p/x.e \rfloor$. If x is a heavy task, then set $x.k = \lfloor x.p/(x.p - x.e) \rfloor$. It is straightforward to prove that consecutive pseudo-deadlines of any task x are either $x.k$ or $x.k + 1$ slots apart.

We define a total order $\sqsupseteq$ on tuples in $\{\mathbf{N}, \{0, +\}, \mathbf{N}\}$ as follows:

$$
\begin{aligned}
&(d_1, s_1, k_1) \sqsupseteq (d_2, s_2, k_2) \iff \\
&\quad (d_1 < d_2) \\
&\quad \bigvee ((d_1 = d_2) \wedge (s_1 = +) \wedge (s_2 = 0)) \\
&\quad \bigvee ((d_1 = d_2) \wedge (s_1 = s_2) \wedge (k_1 \le k_2)).
\end{aligned}
$$

At time t, the total order $\sqsupseteq$ induces an ordering $\trianglerighteq$ on the tasks as follows:

$$
\begin{aligned}
&x \trianglerighteq y \iff \\
&\quad (\delta(x,t), \beta_{\delta(x,t)}(x), x.k) \sqsupseteq (\delta(y,t), \beta_{\delta(y,t)}(y), y.k).
\end{aligned}
$$

The relation $\trianglerighteq$ is essentially a total order over the set of tasks except that $x \trianglerighteq y$ and $y \trianglerighteq x$ may hold for distinct tasks x and y. Such ties may be broken arbitrarily. Hence, in what follows, we treat the relation $\trianglerighteq$ as a total order. For every slot t, Algorithm PD allocates the m resources to the m highest-priority tasks, where priorities are determined as follows (tasks in lower-numbered categories have higher priority):

1. Urgent tasks (all are scheduled).
2. Heavy contending tasks x with $\alpha_{t+1}(x) = +$. Within this category, task x is given priority over task y iff $y \trianglerighteq x$.
3. Light contending tasks x with $\alpha_{t+1}(x) = +$. Within this category, task x is given priority over task y iff $x \trianglerighteq y$.
4. Heavy contending tasks x with $\alpha_{t+1}(x) = 0$.
5. Light contending tasks x with $\alpha_{t+1}(x) = 0$.
6. Remaining heavy contending tasks. Within this category, task x is given priority over task y iff $y \trianglerighteq x$.
7. Remaining light contending tasks. Within this category, task x is given priority over task y iff $x \trianglerighteq y$.

Fix an instance of the periodic scheduling problem, and let $\mathcal{S}_{PD}$ denote the set of all possible schedules that can be produced by Algorithm PD on this instance. In the following, we refer to an arbitrary schedule drawn from this set as S_{PD}.

Theorem 1 *Schedule S_{PD} is P-fair.*

In the remainder of this section, we sketch the proof of Theorem 1. A complete proof appears in [2].

Note that Algorithm PD determines the priority between contending tasks on the basis of the next pseudo-deadlines. In contrast, Algorithm PF takes into account a potentially exponential number of pseudo-deadlines in order to resolve priority.

To prove the correctness of Algorithm PD, we relate its behavior to that of Algorithm PF. We formalize the notion of "good states" in an execution of Algorithm PD in the definition of $G(t)$ below. Informally, a state is good at time t if it is similar to the state reached at time t in an execution of Algorithm PF. It is not the case that every execution of Algorithm PD is always in a good state. However, Algorithm PD closely tracks the behavior of Algorithm PF, in the sense that an execution of Algorithm PD is never in a bad state at two consecutive times t and $t + 1$. To prove this, we first show that an execution of Algorithm PD in a good state at time t has the same number of light urgent tasks and heavy tnegru tasks as an execution of Algorithm PF would have on the same input at time t. Furthermore, even if this execution of Algorithm PD enters a bad state at time $t+1$, it has the same number of light urgent tasks and heavy tnegru tasks as an execution of Algorithm PF would have at time $t+1$. Since a light task that will miss a pseudo-deadline becomes urgent prior to doing so, (and a heavy task becomes tnegru prior to being overallocated), it follows from the correctness of Algorithm PF, and the observation that every execution of Algorithm PD is initially (i.e., at time $t=0$) in a good state, that no execution of Algorithm PD ever misses a pseudo-deadline.

We now show that Algorithm PF is a "specialization" of Algorithm PD, in the sense that the scheduling decisions made by PF from a given state are legal decisions by PD from the same state. In other words, scheduling decisions made by PF are among the possible outcomes that may result from the arbitrary tie-breaking performed within PD.

Lemma 4.1 *The schedule generated by Algorithm* PF *belongs to* $\mathcal{S}_{PD}$.

Now, some definitions and conventions:

- Let $urgent_h(S,t)$ denote the set of all urgent heavy tasks at time t under schedule S; $contending_h(S,t)$, $tnegru_h(S,t)$, $urgent_\ell(S,t)$, $contending_\ell(S,t)$, and $tnegru_\ell(S,t)$ are defined analogously. Furthermore, let $urgent(S,t) = urgent_h(S,t) \cup urgent_\ell(S,t)$ ($contending(S,t)$ and $tnegru(S,t)$ are defined analogously).
- Let x be a light task with its ith pseudo-deadline at slot d. If schedule S allocates a resource to x for the ith time at slot t, we say that *S satisfies x for d at slot t.*
- Let x be a heavy task with its ith pseudo-deadline at slot d. If schedule S does not allocate a resource to x for the ith time at slot t, we say that *S satisfies x for d at slot t.*

For the purposes of understanding Algorithm PD, it may be convenient to view all the pseudo-deadlines of all the light tasks as being arranged in a two-dimensional infinite tableau, with the columns indexed by $(\mathbf{N},\{+,0\})$ pairs, (with the ordering $\cdots < (i-1,+) < (i-1,0) < (i,+) < (i,0) < (i+1,+) < (i+1,0) < \cdots$), and the rows by all integers greater than or equal to 2. (The pseudo-deadlines of the heavy tasks may be viewed analogously, on a separate tableau.) If task x has a pseudo-deadline $+$ (resp., 0) at time t, then this pseudo-deadline "appears" in the row indexed $x.k$, and the column indexed by the ordered pair $(t,+)$ (resp., $(t,0)$). At time t, the priority scheme of Algorithm PD for the light (resp., heavy) tasks then corresponds to allocating (resp., not allocating) the resources in order to satisfy the pseudo-deadlines of urgent (resp., tnegru) and contending tasks in order of increasing column and, within each column, in order of increasing row.

We are now ready to formalize the notion of "good states". For the following definitions assume that $s \in \{+,0\}$.

- $L_s^*(d,k) \stackrel{\text{def}}{=} \{x \in \Gamma \mid x \text{ is light } \wedge x.k = k \wedge \beta_d(x) = s\}$. Also, $\ell_s^*(d,k) \stackrel{\text{def}}{=} |L_s^*(d,k)|$.
- $L_s(S,d,k,t) \stackrel{\text{def}}{=} \{x \in L_s^*(d,k) \mid S \text{ satisfies } x \text{ for } d \text{ strictly prior to slot } t\}$. Also, $\ell_s(S,d,k,t) \stackrel{\text{def}}{=} |L_s(S,d,k,t)|$.
- $H_s^*(d,k)$, $H_s(S,d,k,t)$, $h_s^*(d,k)$ and $h_s(S,d,k,t)$ are defined analogously.
- We say that a given integer-valued expression is *good* if its value is the same for all $S_{PD} \in \mathcal{S}_{PD}$; otherwise, it is *bad*.
- Define the predicate $G_h(t)$ as $(\forall d,k,s \in \{0,+\} :: h_s(S_{PD},d,k,t)$ is good). Define $G_\ell(t)$ similarly, and let $G(t) \stackrel{\text{def}}{=} G_h(t) \wedge G_\ell(t)$. We say that Algorithm PD is *in a good state at time t* if $G(t)$ holds.
- Define the predicate $UCT_h(t)$ as ($|urgent_h(S_{PD},t)|$, $|contending_h(S_{PD},t)|$, and $|tnegru_h(S_{PD},t)|$ are good). Define $UCT_\ell(t)$ similarly, and let $UCT(t) \stackrel{\text{def}}{=} UCT_h(t) \wedge UCT_\ell(t)$.

(Observe that $G(0)$ and $UCT(0)$ hold trivially, i.e., the initial state is good.)

A "good" state is therefore one in which all PD-generated schedules (i.e., all $S_{PD} \in \mathcal{S}_{PD}$, and specifically, the schedule generated by Algorithm PF) have the same number of not-yet-satisfied pseudo-deadlines in each cell of the two tableaux referred to above. (Note that we do not require that these pseudo-deadlines belong to the same tasks in all schedules, but only that their number be the same.) We prove that all schedules $S_{PD} \in \mathcal{S}_{PD}$ allocate the same number of resources to light tasks and to heavy tasks in good states. We then show that all schedules S_{PD} are in a good state at time $(t+1)$ as well, unless one of a few very special scenarios is encountered. One such scenario is outlined in the next paragraph; the remaining scenarios which lead to a bad state are virtually identical in structure.

Assume Algorithm PD is in a "good" state at time t. Consider two light (resp., heavy) tasks x and y such that: (i) $x.k = y.k = k$, (ii) $\beta_t(x) = \beta_t(y) = +$, (iii) $\delta(x,t) = t+k$, and (iv) $\delta(y,t) = t+k+1$. Let S_1 and S_2 be two schedules satisfying the following properties at time t: (i) schedule S_1 has already satisfied the pseudo-deadline $\beta_t(x)$, but not yet the pseudo-deadline $\beta_t(y)$, (ii) schedule S_2 has already satisfied the pseudo-deadline $\beta_t(y)$ of y, but not yet the pseudo-deadline $\beta_t(x)$. Thus, in schedule S_1, y is urgent (resp., tnegru) and x is contending, whereas in schedule S_2, x is urgent (resp., tnegru) and y is contending. Note that the given scenario does not violate the assumption that $G(t)$ holds. At slot t, however, it is now possible that schedules S_1 and S_2 both satisfy task x and task y, but for different pseudo-deadlines: Schedule S_1 may satisfy task y for pseudo-deadline $\beta_t(y)$ and task x for pseudo-deadline $\beta_{t+k}(x)$, while S_2 may satisfy task x for pseudo-deadline $\beta_t(x)$ and task y for pseudo-deadline $\beta_{t+k+1}(y)$. In S_2, the pseudo-deadline $\beta_{t+k}(x)$ of x has higher priority than the pseudo-deadline $\beta_{t+k+1}(y)$ of y, but no task may be allocated multiple copies of the resource in the same slot. Therefore we have reached a "bad" state; $G(t+1)$ does not hold.

In the proof, most of the technical lemmas are devoted to a study of the special case where $G(t)$ holds, but $G_\ell(t+1)$ (resp., heavyGoodt+1) does not. (We prove that at least one of $G_\ell(t+1)$ or $G_h(t+1)$ holds whenever $G(t)$ holds.) If this is the case, we show that there are no light (resp., heavy) urgent (resp., tnegru) tasks at time $t+1$. This in turn implies that no scenario similar to that described in the preceding paragraph can occur at time $t+1$; note that the above scenario requires one of tasks x and y to be urgent (resp., tnegru) in each of S_1 and S_2. We then show that, despite the fact that we have reached a bad state at time $t+1$, $UCT(t+1)$ holds and we return to a good state at time $t+2$.

Theorem 2 *Algorithm* PD *can be implemented in* $O(\min(n, m \log n))$ *time.*

Proof: In conjunction with any linear-comparison selection algorithm (e.g., [4]), the constant time priority comparison algorithm described in this section provides an $O(n)$ time implementation of Algorithm PD. A heap-based implementation that runs in $O(m \log n)$ time is described in Appendix A. □

5 Conclusions

The techniques presented in this paper build on the results established in [1]. As argued in the introduction, the tie-breaking procedure of [1] may be viewed as a natural generalization of an "earliest deadline" strategy. As the earliest deadline paradigm has proven to be useful for solving a large number of scheduling problems, especially problems involving a single resource (such as uniprocessor scheduling problems), it seems likely that the P-fairness-based approach of [1] will be useful for solving an even larger class of scheduling problems, especially problems involving multiple resources.

The importance of the work in the present paper is that it demonstrates an approach for obtaining highly efficient scheduling algorithms based on P-fairness. More specifically, our work suggests the following general plan for attacking a given scheduling problem:

1. Prove that a P-fair solution exists for the problem in question. In the case of the periodic scheduling problem, this step was accomplished using a network flow argument [1, Theorem 1].
2. Find the "canonical" P-fair algorithm for solving the problem. In the case of the periodic scheduling problem, this is Algorithm PF of [1]. Informally, we view Algorithm PF as the canonical P-fair scheduling algorithm for the periodic scheduling problem because every scheduling decision that it makes is "locally optimal" with respect to preserving P-fairness. This local optimality is achieved by looking arbitrarily far into the future when making current scheduling decisions.
3. Define a "limited lookahead" version of the canonical P-fair algorithm, and prove that the behavior of this algorithm closely tracks that of the canonical P-fair algorithm. In the case of the periodic scheduling problem, Algorithm PD plays the role of the limited lookahead algorithm.
4. Prove that the limited lookahead algorithm can be implemented efficiently. Because Algorithm PD uses only constant lookahead, this analysis is completely straightforward in the case of the periodic scheduling problem.

Note that the "canonical" P-fair algorithm of Step 2 above may be extremely slow. For example, in establishing the correctness of Algorithm PD, we have relied solely on the correctness of Algorithm PF; we have not relied on the existence of an efficient (e.g., polynomial time) implementation of Algorithm PF.

References

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

[2] S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. Fast scheduling of periodic tasks on multiple resources. Technical Report TR–95–02, Department of Computer Science, University of Texas at Austin, February 1995.

[3] S. K. Baruah, R. R. Howell, and L. E. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2:301–324, 1990.

[4] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7:448–461, 1973.

[5] W. A. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21:177–185, 1974.

[6] C. L. Liu. Scheduling algorithms for multiprocessors in a hard-real-time environment. JPL Space Programs Summary 37–60, vol. II, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, pages 28–37, November 1969.

[7] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *JACM*, 20:46–61, 1973.

[8] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21:309–315, 1978.

A An $O(m \lg n)$ Implementation

Algorithm PD may be efficiently implemented by using the binomial heap data structure of Vuillemin [8]. The binomial heap supports the operations listed below, as well as certain others that do not concern us here. (In the table below, H, H_0, and H_1 are of type binomial heap, x is a heap element, and S denotes a set of n heap elements.)

Operation	Worst-Case Complexity
$H :=$ MakeHeap()	$O(1)$
$H :=$ BuildHeap(S)	$O(n)$
Insert(H, x)	$O(\lg n)$
$x :=$ ExtractMin(H)	$O(\lg n)$
$H :=$ Union(H_0, H_1)	$O(\lg n)$

For each task x we maintain a record with the following information: (i) $x.e$, $x.p$, and $x.k$, which contain fixed integer values, (ii) the number of times task x has been scheduled in its current period, and (iii) the number of slots remaining until the end of the current period. For the sake of efficiency, values (ii) and (iii) are not updated at each time step but are only generated as necessary. Other important quantities such as the lag of task x, the number of slots until the next pseudo-deadline of x, or the symbol associated with the next pseudo-deadline of x (i.e., $+$ or 0), can be easily determined from the aforementioned values using a constant number of integer operations.

Our implementation uses a number of binomial heaps: (i) H, which stores the task requests that are currently eligible to be scheduled (i.e., those that are not tnegru), and (ii) various heaps $H_{t'}$, for times t' when some currently tnegru task will become contending or urgent. Since there are n tasks, there will be no more than $n+1$ non-empty binomial heaps at any given time. The relative priorities of the tasks in each heap are determined using the seven-level scheme given in Section 4.

The pseudo-code for the implementation is given in Figure 1. Each iteration of the main loop in Lines (2) to (12) corresponds to the scheduling of one time slot. The "pre-processing" of Line (1) takes $O(n)$ time. (Note that we do not include this pre-processing time in the per slot time complexity of our algorithm.) Within each iteration of the main loop, the **repeat** loop of Lines (3) to (8) is executed m times. Line (4) requires $O(\lg n)$ time, and Lines (5) to (6) require $O(1)$ time. We argue below that Lines (7) and (9) can each be implemented to run in $O(\lg n)$ time. Line (10) also takes $O(\lg n)$ time, and so the overall complexity of Algorithm PD is $O(m \lg n)$.

To efficiently execute the test in Line (1) of procedure Requeue, we maintain another search structure (e.g., any standard dictionary data structure such as a red-black tree) that contains pointers to each of these binomial heaps (binomial heap H_t is indexed by t), and that permits $O(\lg n)$-time implementations of the operations Insert, Delete, and Find. Lines (1) to (3) of Requeue therefore take $O(\lg n)$ time, and Line (5), which is an insertion into a binomial heap, also takes $O(\lg n)$ time. Line (9) of PD is implemented as a Find in the dictionary, followed by a Delete if necessary; each of these operations runs in $O(\lg n)$ time.

```
Algorithm PD
begin
    H := BuildHeap(Γ);
    for t := 0, 1, 2, ... do
        repeat
            x := ExtractMin(H);
            "Schedule task x in slot t"
            t' := "the earliest future time at
                    which task x will not be
                    tnegru";
            Requeue(x, t')
        until "m tasks have been scheduled
                in slot t";
        if "Heap H_{t+1} exists" then
            H := Union(H, H_{t+1})
        fi
    od
end
```

```
Requeue(x, t)
begin
    if "Heap H_t does not exist" then
        H_t := MakeHeap()
    fi;
    "Update fields associated with task x (e.g.,
            the number of allocations in the
            current period)";
    Insert(H_t, x)
end
```

Figure 1: Algorithm PD.