Fair On-Line Scheduling of a Dynamic Set of Tasks on a Single Resource ⋆

Sanjoy K. Baruah [1]

Department of Computer Science, University of Vermont, Burlington, VT 05405. E-mail: sanjoy@cs.uvm.edu.

Johannes E. Gehrke

Computer Sciences Department, University of Wisconsin–Madison, 1210 West Dayton Street, Madison, WI 53706–1685. E-mail: johannes@cs.wisc.edu.

C. Greg Plaxton [2]

Department of Computer Science, University of Texas at Austin, Austin, TX 78712–1188. E-mail: plaxton@cs.utexas.edu.

Ion Stoica

Department of Computer Science, Old Dominion University, Norfolk, VA 23529–0162. E-mail: stoica@cs.odu.edu.

Hussein Abdel-Wahab

Department of Computer Science, Old Dominion University, Norfolk, VA 23529–0162. E-mail: wahab@cs.odu.edu.

Kevin Jeffay

Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, NC 27599–3175. E-mail: jeffay@cs.unc.edu.

Abstract

Consider a set of "tasks" competing for the use of a single "resource", where: (i) only one task is allowed to use the resource at a time, (ii) the resource is scheduled in unit-time intervals, (iii) each task requires a specific fraction of the resource capacity over an extended period, and (iv) tasks arrive and depart at any time. We refer to such a task system as an instance of the *single-resource scheduling problem*. The problem of designing a "fair" scheduling algorithm for such task systems has recently received a great deal of attention in the literature. This paper makes two main contributions. First, we point out that Tijdeman's work on the so-called "chairman assignment problem" provides a simple and efficient on-line algorithm for the static version of the single-resource scheduling problem (i.e., where the set of tasks competing to use the resource does not change over time). We then extend Tijdeman's algorithm to obtain a simple and efficient on-line algorithm for the dynamic single-resource scheduling problem.

Keywords: algorithms, fairness, on-line scheduling, periodic tasks.

---

1 Introduction

In many computer systems, various "resources" (e.g., processors, disks, network bandwidth) may be shared among different "tasks" (e.g., user processes), where each task requires access to a particular shared resource at a steady rate for an extended period of time. For example, a network link may support a number of real-time data streams with each data stream requesting a certain bandwidth. Or, a movie-on-demand server may retrieve several programmes concurrently from storage disks with a limited number of read heads. Or, a processor may support a number of long-term processes, each of which requires a certain fraction of the processor capacity.

In such instances, the resource is often characterized by the property that it is assigned to a task for integer multiples of some basic time unit. For a fixed-size packet network, this unit might correspond to the time required to send a single packet. For disk access, it is determined by such factors as the size of a disk block and the seek and scan times, while for shared CPUs a scheduling quantum might correspond to a fixed number of clock cycles. This property is formalized as the *integer boundary constraint*: Time is viewed as being divided into a countably infinite number of equal-sized "slots", numbered from 0, with each slot corresponding to one basic time unit. The resource is allocated to exactly one task (or remains unallocated) during each slot.

Suppose that a given task X requires a fraction w, $0 < w < 1$, of a shared resource over an extended period of time. Ideally, the resource would be assigned to X in $w \cdot t$ of the first t slots, for all t. However, this is clearly impossible due to the integer boundary constraint; for example, after one time slot the resource will have been assigned to X either 0 or 1 times, and not w times. The best we can hope to do is to assign the resource to X in either $\lfloor w \cdot t \rfloor$ or $\lceil w \cdot t \rceil$ of the first t slots, for all t. A schedule that simultaneously provides such *proportionate progress* to all tasks is said to be *P-fair* [1]. More generally, a schedule for which the number of allocations to any task is at all times within an additive Δ of

---

[⋆] Related results appear in the conference paper [14] as well as the technical reports [2,12].

[1] Supported by the National Science Foundation under Research Initiation Award No. CCR–9596282.

[2] Supported by the National Science Foundation under Grant No. CCR–9504145, and the Texas Advanced Research Program under Grant No. ARP–93–00365–461.

the ideal value is said to achieve a *lag bound* of Δ. (Note that a P-fair schedule achieves a lag bound of less than 1.) The problem of designing an efficient on-line algorithm for single-resource scheduling that achieves a small lag bound has recently received a great deal of attention in the literature [8,13,16–18]. In the context of scheduling network traffic in packet-based networks, Demers, Keshav, and Shenker [7] have proposed the *Packet Fair Queueing algorithm* (PFQ). Parekh and Gallager [10,11] have analyzed the behavior of PFQ when the traffic conforms to the (σ, ρ) model [6], and shown that any task X receives the resource for at least $\lfloor w \cdot t \rfloor$ of the first t slots; it has been shown, however [4,12,16], that X may in fact receive the resource for $w \cdot t + \Theta(n)$ of the first t slots, where n denotes the number of tasks sharing the resource. Thus, none of the algorithms presented in the aforementioned papers achieves a constant lag bound (i.e., independent of n, the number of tasks), let alone P-fairness. In this paper, we extend a result of Tijdeman [15] to obtain a simple P-fair algorithm for scheduling a dynamic set of tasks on a single resource.

When the cumulative request of all tasks of a resource exceeds the capacity of the resource, at least two different strategies are possible. One strategy is to identify a maximal subset of the requesting tasks whose requests can all be accommodated, and to deny service to the rest. Another strategy, the one adopted in this research, is to offer every task a fraction of its requested capacity, the fraction offered depending upon the degree of overload. (Thus, if all the requests made of a resource of capacity C sum to R, $R > C$, each task is offered a fraction C/R of its requested capacity.) This strategy reacts to slight overload by offering a slightly degraded quality of service to all tasks, rather than choosing to maintain optimal service to some tasks while providing no service to others. The task semantics determine what to do with this reduced capacity. For example, a process representing an MPEG stream of video data can tolerate a certain amount of slowdown that is indiscernible to the human eye. On the other hand, an audio stream may choose to discard some data, and maintain the rate of data delivery. (A task that does not wish to accept this lesser level of service can, of course, decline to do so.)

The remainder of this paper is organized as follows. In Section 2, we define some basic terminology and formulate several variants of the single-resource scheduling problem. In Section 3, we discuss certain consequences of Tijdeman's work on the so-called "chairman assignment problem" [15], including an efficient on-line scheduling algorithm for the static version of the single-resource scheduling problem. In Section 4, we state our main result, an efficient on-line scheduling algorithm for a dynamic version of the single-resource scheduling problem. In Section 5, we present this algorithm, prove its correctness, and discuss the details of an efficient implementation. In Section 6, we offer some concluding remarks.

2 Terminology

We now define a number of terms to be used in the description and analysis of the scheduling problems addressed in this paper.

We have a (possibly infinite) set of *tasks* Γ competing for the use of a single *resource* that can accommodate only one task at a time. The resource is available to be scheduled (i.e., assigned to some specific task, or to no task) during a (possibly infinite) number of non-overlapping intervals of time that we refer to as *slots*. The slots are numbered from 0.

Let $\Gamma_t \subseteq \Gamma$ denote the finite set of tasks that are available to be scheduled in slot t. For $t > 0$, we view the set Γ_t as being obtained from Γ_{t-1} by *inserting* the tasks in $\Gamma_t \setminus \Gamma_{t-1}$ and *deleting* the tasks in $\Gamma_{t-1} \setminus \Gamma_t$. If a particular task x is deleted at slot t, we do not allow x to be inserted at any later slot $t' > t$. If $\Gamma_t = \Gamma$ for all t, we say that the set of tasks is *static*. Otherwise, it is *dynamic*.

A *schedule* S is a function from $\mathbf{N}$ to $\Gamma \cup \{\#\}$ such that $S(t)$ is an element of $\Gamma_t \cup \{\#\}$ for all t. Schedule S is interpreted as follows: (i) if $S(t) = x$ where x is in Γ_t, then task x is assigned to the resource in slot t, and (ii) if $S(t) = \#$, then no task is assigned to the resource in slot t.

A *scheduling algorithm* computes a schedule S by successively computing $S(0)$, $S(1)$, $S(2)$, and so on. A scheduling algorithm has *preprocessing cost* $a(n)$, *per-slot cost* $b(n)$, *insertion cost* $c(n)$, and *deletion cost* $d(n)$ if: (i) the time to compute $S(0)$ is $O(a(n) + b(n))$ where $n = |\Gamma_0|$, (ii) for all $t > 0$ the time to compute $S(t)$ (excluding the time to compute $S(0), \ldots, S(t-1)$) is $O(b(n) + x \cdot c(n) + y \cdot d(n))$ where $x = |\Gamma_t \setminus \Gamma_{t-1}|$, $y = |\Gamma_{t-1} \setminus \Gamma_t|$, and $n = \max\{|\Gamma_{t-1}|, |\Gamma_t|\}$.

For each slot t and each task x in Γ there is an associated nonnegative real number $\mathsf{r}(x,t)$, which we refer to as the *request* of task x at slot t. If x is not in Γ_t then $\mathsf{r}(x,t) = 0$. We say that the task requests are *constant* if for each task x, and for any pair of slots t and t' such that $x \in \Gamma_t \cap \Gamma_{t'}$, $\mathsf{r}(x,t) = \mathsf{r}(x,t')$. Otherwise, we say that the task requests are *variable*.

The *total request* at slot t, denoted $\mathsf{R}(t)$, is the sum over all x in Γ of $\mathsf{r}(x,t)$. The *scaling factor* at slot t, denoted $\mathsf{f}(t)$, is 1 if $\mathsf{R}(t) \leq 1$, and $1/\mathsf{R}(t)$ otherwise. The *weight* of task x at slot t, denoted $\mathsf{w}(x,t)$, is $\mathsf{r}(x,t) \cdot \mathsf{f}(t)$. Note that $0 \leq \mathsf{w}(x,t) \leq 1$, and that the sum of the task weights at any given slot also lies in the real interval $[0,1]$.

In applications, $\mathsf{r}(x,t)$ will typically be less than 1, and should be interpreted as the fraction of the resource that task x would ideally like to receive during slot t. Unfortunately, each slot is indivisible so a scheduling algorithm cannot assign the resource to task x for a non-trivial fraction of a slot. Even if the slots were divisible, a scheduling algorithm could not hope to satisfy the requests of all tasks at slot t unless $\mathsf{R}(t) \leq 1$; the task weights $\mathsf{w}(x,t)$ should be viewed as adjusted (in a fair manner) task requests that take this observation into account. In particular, note that $\mathsf{w}(x,t) = \mathsf{r}(x,t)$ if $\mathsf{R}(t) \leq 1$ (i.e., the system is not overloaded and no adjustment is necessary), and $\mathsf{w}(x,t) = \mathsf{r}(x,t)/\mathsf{R}(t)$ if $\mathsf{R}(t) > 1$ (i.e., the system is overloaded and so we uniformly scale down all the task requests to obtain a set of corresponding task weights that sums to 1).

As indicated above, we still need to address the indivisibility of the slots. To do so, we introduce a couple of additional definitions. For each task x and slot t, let

$$\mathsf{W}(x,t) = \sum_{0 \le t' < t} \mathsf{w}(x,t').$$

Informally, $\mathsf{W}(x,t)$ represents the "ideal" number of times for task x to be assigned the resource in slots 0 through $t-1$. It will prove to be useful to generalize the preceding definition of $\mathsf{W}(x,t)$ to allow for arbitrary nonnegative real values of t. However, for the sake of clarity, we prefer to reserve the variable t to denote only integer slots; the variable τ will be used to denote real slots, as in the following definition.

$$\mathsf{W}(x,\tau) = \left(\sum_{0 \le t < \lfloor \tau \rfloor} \mathsf{w}(x,t) \right) + (\tau - \lfloor \tau \rfloor) \cdot \mathsf{w}(x, \lfloor \tau \rfloor).$$

For any given schedule S, task x, and slot t, let $\mathsf{A}(S,x,t)$ denote the number of times that task x is allocated the resource in slots 0 through $t-1$ under schedule S, and define the lag of task x at slot t with respect to schedule S as

$$\mathsf{Lag}(S,x,t) = \mathsf{W}(x,t) - \mathsf{A}(S,x,t).$$

Because of the indivisibility of slots we cannot hope to maintain a lag of 0 for all tasks at all slots. Instead, we focus on the design of schedules S for which

$$\max_{x,t} |\mathsf{Lag}(S,x,t)|$$

is as small as possible.

A *lag bound* L is a pair $(\prec_L, \Delta_L)$ where $\prec_L$ is either $<$ or $\le$, and Δ_L is a real number. The results of the present paper are concerned with lag bounds of the form $(<, 1)$ and $(\le, 1 - 1/(2n-2))$, where n is a given upper bound on $\max_t |\Gamma_t|$. For the sake of brevity, we will often refer to these lag bounds as < 1 and $\le 1 - 1/(2n-2)$, respectively. A scheduling algorithm *achieves a lag bound* of L if and only if any schedule S produced by the algorithm satisfies $|\mathsf{Lag}(S,x,t)| \prec_L \Delta_L$ for all tasks x and slots t. A schedule or scheduling algorithm that achieves a lag bound of < 1 is said to be *P-fair* [1]. Note that P-fairness is a very strong fairness property: A schedule S is P-fair if and only if for all tasks x and slots t, either $\mathsf{A}(S,x,t) = \lfloor \mathsf{W}(x,t) \rfloor$ or $\mathsf{A}(S,x,t) = \lceil \mathsf{W}(x,t) \rceil$. All of the algorithms described in this paper are P-fair.

For any schedule S, task x, slot t, and lag bound L, let the predicate $\mathsf{Contending}(S,x,t,L)$ be defined as

$$-\Delta_L \prec_L \mathsf{Lag}(S,x,t) + \mathsf{w}(x,t) - 1. \tag{1}$$

Note that $\mathsf{Contending}(S,x,t,L)$ holds if and only if task x could be scheduled in slot t without violating the lower bound on lag associated with L. We also define $\mathsf{Deadline}(S,x,t,L)$ as the (least, if $\prec_L$ is $<$, and greatest, if $\prec_L$ is $\le$) real slot τ such that

$$\mathsf{Lag}(S, x, t) + \mathsf{W}(x, \tau) - \mathsf{W}(x, t) = \Delta_L.$$

Note that if $\prec_L$ is $<$ (resp., $\leq$), then task x must be scheduled in at least one of the slots $\{t, t+1, \ldots, \lceil \tau \rceil - 1\}$ (resp., $\{t, t+1, \ldots, \lfloor \tau \rfloor\}$) in order to avoid violating the upper bound on lag associated with L.

A scheduling algorithm is *on-line* if for all slots t it computes $S(t)$ without any knowledge of: (i) future task sets (i.e., $\Gamma_{t'}$ for $t' > t$), in cases where the set of tasks is dynamic, and (ii) future task requests (i.e., $\mathsf{r}(x, t')$ for $t' > t$), in cases where the task requests are variable. Otherwise, it is *off-line*.

In this paper, we discuss the complexity (in terms of preprocessing cost, per-slot cost, insertion cost, and deletion cost) of both on-line and off-line scheduling algorithms achieving lag bounds of < 1 or better. We consider the following specific variations of the basic scheduling problem discussed above:

- Problem A: Static task set, constant task requests.
- Problem B: Dynamic task set, constant task requests, given upper bound n on $\max_t |\Gamma_t|$.
- Problem C: Dynamic task set, constant task requests.
- Problem D: Dynamic task set, variable task requests, given upper bound n on $\max_t |\Gamma_t|$.
- Problem E: Dynamic task set, variable task requests.

Note that we do not consider the case of a static task set with variable task requests; that case is equivalent to Problem D. We further remark that the above list of problems is arranged in increasing order of difficulty, except that Problems C and D are incomparable. (It turns out that Problem C is easier than Problem D in the on-line setting, which is our main interest, but that a slightly better lag bound is achievable for Problem D in the off-line setting.) Thus: (i) any upper bound on the complexity of Problem B also applies to Problem A, (ii) any upper bound on the complexity of Problem C or D also applies to Problems A and B, and (iii) any upper bound on the complexity of Problem E also applies to Problems A, B, C, and D.

3 Previous Results

Work of Tijdeman [15] on the so-called "chairman assignment problem" has direct implications for the problems considered in this paper.

Lemma 1 (Tijdeman) *There exists a schedule with lag bound $\leq 1 - 1/(2n-2)$ for any instance of Problem D (hence also for Problems A and B). Furthermore, there exists a schedule with lag bound < 1 for any instance of Problem E (hence also for Problem C).*

The following scheme is implicit in [15], and underlies a number of scheduling algorithms for Problems A through E.

Tijdeman's Scheme. We wish to generate a schedule for a given instance of Problem A, B, C, D, or E, subject to a given lag bound L that is known to be achievable by Lemma 1. Assume that $S(0)$ through $S(t-1)$ have already been computed, $t \geq 0$. We now compute $S(t)$ as follows. First, define task x to be *contending* if and only if $\mathsf{Contending}(S, x, t, L)$ holds. Next, define the *deadline* of each contending task x as $\mathsf{Deadline}(S, x, t, L)$. If there are no contending tasks, set $S(t) = \#$; otherwise, set $S(t)$ to any earliest-deadline task x.

We have chosen to refer to the above procedure as a "scheme", and not an "algorithm", because in general it is not possible to calculate the task deadlines in a finite number of steps. For example, if the task requests are variable and the length of the schedule being computed is infinite, it may be necessary to examine an infinite number of future task requests in order to compute even a single task deadline. (By contrast, it is easy to determine the set of contending tasks, even in an on-line sense.) On the other hand, in cases where the task deadlines can be computed in a finite number of steps (e.g., if the length of the schedule is finite), Tijdeman's Scheme provides an off-line algorithm.

Theorem 1 (Tijdeman) *Restricting attention to problem instances that admit an off-line algorithm for computing the task deadlines, there is an off-line algorithm for Problems A, B, and D (resp., C and E) with lag bound* $\leq 1 - 1/(2n-2)$ *(resp.,* < 1*).*

Of course, if we could give an on-line algorithm for computing the task deadlines, then Tijdeman's Scheme could also be implemented on-line. Unfortunately, it is easy to see that the task deadlines cannot possibly be computed on-line for general instances (even finite instances) of Problems B, C, D, and E. On the other hand, for instances of Problem A, it is easy to compute task deadlines on-line; because the task requests are constant and the set of tasks is static, $\mathsf{w}(x, t)$ does not depend on t. Tijdeman [15] gives the following on-line algorithm for Problem A with $L = (\leq, 1 - 1/(2n-2))$, and where we write $\mathsf{w}(x)$ to denote $\mathsf{w}(x, t)$.

Algorithm A. Proceed as in Tijdeman's Scheme above, but calculate $\mathsf{Deadline}(S, x, t, L)$ using the formula $t + \Lambda$ where

$$\Lambda = \frac{\Delta_L - \mathsf{Lag}(S, x, t)}{\mathsf{w}(x)}.$$

The correctness of Algorithm A follows immediately from the correctness of Tijdeman's Scheme. Note that Algorithm A remains correct if we replace the deadline $t + \Lambda$ with Λ ("slack"). (In fact, the algorithm given in [15] makes use of Λ instead of $t + \Lambda$.)

A naive implementation of Algorithm A leads to a preprocessing cost of $O(n)$ and a per-slot cost of $O(n)$. Using standard algorithmic techniques (see, for example, the implementation of Algorithm PD in [3]) these bounds can be improved to obtain the following result. (Although no implementation details are provided in [15], we attribute the result to Tijdeman since these details are straightforward.)

Theorem 2 (Tijdeman) *Problem A can be solved by an on-line algorithm with lag bound* $\leq 1 - 1/(2n-2)$*, preprocessing cost* $O(n)$*, and per-slot cost* $O(\log n)$*.*

4 Our Results

In this paper we show that Tijdeman's Scheme admits an efficient on-line implementation for solving Problems B and C with small lag bounds. We refer to this implementation as Algorithm BC. Interestingly, these results are achieved in spite of our earlier observation (Section 3) that task deadlines cannot be computed on-line. The main idea underlying Algorithm BC is that it is possible to compute on-line a "virtual" deadline for each task such that the relative order of the virtual deadlines is the same as the relative order of the (unknown) deadlines. Thus, virtual deadlines can be used instead of deadlines within Tijdeman's Scheme.

Theorem 3 *Problem B (resp., Problem C) can be solved by an on-line algorithm with lag bound* $\leq 1 - 1/(2n-2)$ *(resp.,* < 1*), preprocessing cost* $O(n \log n)$ *(resp.,* $O(n)$*), per-slot cost* $O(\log n)$*, insertion cost* $O(\log n)$*, and deletion cost* $O(\log n)$*, under the assumption that no task is ever deleted when its lag is negative.*

The technical assumption regarding deletion in the statement of Theorem 3 may seem somewhat artificial, but cannot be dropped while maintaining a lag bound of < 1 or better. For example, consider an instance of Problem B with 10 tasks where the request of each task is 0.1 at each slot. If no tasks are inserted or deleted in the first 8 slots, then some pair of tasks x and y will have lag 0.8 at $t = 8$ (since at least two tasks have not been scheduled in any of the first 8 slots). If the other 8 tasks are deleted at $t = 8$, then at least one of x and y will have lag 1.3 at $t = 9$ (whichever one is not scheduled in slot 8).

In practice, our assumption regarding deletion should not pose any serious concern, since the deletion of a task with negative lag can simply be "delayed" until the lag of that task reaches 0. (Note that the lag of a task increases as long as the task is not scheduled, and so we only need to ensure that the scheduling algorithm does not assign the resource to a "deleted" task.)

5 An On-Line Algorithm for Problems B and C

In this section we prove Theorem 3 by giving an on-line algorithm for Problems B and C, along with an efficient implementation of the algorithm. The algorithm is parameterized by a lag bound L which should be set to $\leq 1 - 1/(2n-2)$ for Problem B, and to < 1 for Problem C.

Since Problems B and C involve constant task requests, we write $\mathsf{r}(x)$ instead of $\mathsf{r}(x,t)$ throughout this section.

With any real slot τ we associate a virtual slot $\mathsf{v}(\tau)$, defined as

$$\mathsf{v}(\tau) = \left(\sum_{0 \le t < \lfloor \tau \rfloor} \mathsf{f}(t)\right) + (\tau - \lfloor \tau \rfloor) \cdot \mathsf{f}(\lfloor \tau \rfloor).$$

Lemma 2 *For any real slots τ and τ', $\tau \le \tau'$ if and only if $\mathsf{v}(\tau) \le \mathsf{v}(\tau')$.*

Proof: Straightforward since $\mathsf{f}(t) \ge 0$ for all slots t. ■

Lemma 3 *For any task x and real slot τ, we have*

$$\mathsf{r}(x) \cdot \mathsf{v}(\tau) = \mathsf{W}(x, \tau).$$

Proof: Straightforward since $\mathsf{r}(x) \cdot \mathsf{f}(t) = \mathsf{w}(x,t)$. ■

We can assume that our scheduling algorithm initially discards all tasks x with $\mathsf{r}(x) = 0$, since such tasks never need to be scheduled. Thus, in the pair of definitions that follow, we can divide by $\mathsf{r}(x)$ without worrying about dividing by zero.

For any schedule S, task x, slot t, and lag bound L, we define

$$\mathsf{VirtualReleaseSlot}(S, x, t, L) = \frac{1 - \Delta_L + \mathsf{A}(S, x, t)}{\mathsf{r}(x)}$$

and

$$\mathsf{VirtualDeadline}(S, x, t, L) = \frac{\Delta_L + \mathsf{A}(S, x, t)}{\mathsf{r}(x)}.$$

Lemma 4 *For any schedule S, task x, slot t, and lag bound L, $\mathsf{Contending}(S,x,t,L)$ holds if and only if*

$$\mathsf{VirtualReleaseSlot}(S, x, t, L) \prec_L \mathsf{v}(t) + \mathsf{f}(t).$$

Proof: Multiplying both sides of the given inequality by $\mathsf{r}(x)$, we obtain

$$1 - \Delta_L + \mathsf{A}(S, x, t) \prec_L \mathsf{W}(x, t) + \mathsf{w}(x, t),$$

which is logically equivalent to

$$\begin{aligned} -\Delta_L &\prec_L \mathsf{W}(x,t) - \mathsf{A}(S,x,t) + \mathsf{w}(x,t) - 1 \\ &= \mathsf{Lag}(S,x,t) + \mathsf{w}(x,t) - 1, \end{aligned}$$

and hence also to $\mathsf{Contending}(S, x, t, L)$. ■

Lemma 5 *For any schedule S, task x, slot t, and lag bound L, we have*

$$\mathsf{VirtualDeadline}(S,x,t,L) = \mathsf{v}(\mathsf{Deadline}(S,x,t,L)).$$

Proof: Let $\mathsf{Deadline}(S,x,t,L) = \tau$. Thus

$$\mathsf{Lag}(S,x,t) + \mathsf{W}(x,\tau) - \mathsf{W}(x,t) = \Delta_L$$

and hence

$$\begin{aligned}\mathsf{VirtualDeadline}(S,x,t,L) &= \frac{\mathsf{Lag}(S,x,t) + \mathsf{W}(x,\tau) - \mathsf{W}(x,t) + \mathsf{A}(S,x,t)}{\mathsf{r}(x)} \\ &= \frac{\mathsf{W}(x,\tau)}{\mathsf{r}(x)} \\ &= \mathsf{v}(\tau),\end{aligned}$$

where the last equation follows from Lemma 3. ■

Lemma 6 *For any schedule S, tasks x and y, slot t, and lag bound L, we have*

$$\mathsf{VirtualDeadline}(S,x,t,L) \le \mathsf{VirtualDeadline}(S,y,t,L)$$

if and only if

$$\mathsf{Deadline}(S,x,t,L) \le \mathsf{Deadline}(S,y,t,L).$$

Proof: Immediate from Lemmas 2 and 5. ■

Algorithm BC. Proceed as in Tijdeman's Scheme, but use $\mathsf{VirtualDeadline}(S,x,t,L)$ instead of $\mathsf{Deadline}(S,x,t,L)$.

The correctness of Algorithm BC follows immediately from the correctness of Tijdeman's Scheme and Lemma 6. It remains to establish the time bounds claimed in Theorem 3. A straightforward implementation of Algorithm BC results in an $O(n)$ per-slot cost.

In order to obtain an efficient implementation of Algorithm BC, we make use of an abstract data structure for maintaining a dynamic set X of triples $T = (T.x, T.a, T.b)$, where $T.x$ is a task and $T.a$ and $T.b$ are real numbers. The set X contains at most one entry associated with any particular task at any given time, that is, if T and T' belong to X then either $T.x \neq T'.x$ or $T = T'$. For any real number a and lag bound L, let $\mathsf{Triples}(X,a,L)$ denote the set of all T in X such that $T.a \prec_L a$, and let $\mathsf{MinTriple}(X,a,L)$ denote: (i) $\#$, if $\mathsf{Triples}(X,a,L)$ is empty, and (ii) $T.x$ for some triple T in $\mathsf{Triples}(X,a,L)$ such that $T.b \le T'.b$ for all T' in $\mathsf{Triples}(X,a,L)$, if $\mathsf{Triples}(X,a,L)$ is non-empty.

The operations allowed on the set X are as follows: (i) $\mathsf{Insert}(T)$, which is applicable only if T is a triple satisfying $T.x \neq T'.x$ for all T' in X, and which inserts T into the set X; (ii) $\mathsf{Delete}(x)$, which is applicable only if there is a (unique) triple T in X such that $T.x = x$, and which returns and deletes the triple T from the set X; (iii) $\mathsf{2D\text{-}FindMin}(a, L)$, which is applicable for any real number a and lag bound L, and which returns $\mathsf{MinTriple}(X, a, L)$. All three of these operations can easily be implemented to run in worst-case $O(\log |X|)$ time using an appropriately augmented red-black tree data structure [5,9].

Given the aforementioned data structure, we implement Algorithm BC as follows. At slot 0, we set $\mathsf{v}(0) = 0$ and compute $\mathsf{f}(0)$ (at a cost that is linear in $|\Gamma_0|$). For each task x in Γ_0, we perform an $\mathsf{Insert}(T)$ operation to add the triple

$$T = (x, \mathsf{VirtualReleaseSlot}(S, x, 0, L), \mathsf{VirtualDeadline}(S, x, 0, L))$$

to the (initially empty) dynamic set X. We remark that the cost of these insertions is $O(n \log n)$ where $n = |\Gamma_0|$, but that this cost can be reduced to $O(n)$ if $L = (<, 1)$, because in that case all of the virtual release slots are initially equal to 0. (The underlying red-black tree is ordered by virtual release slot.)

Having properly initialized the set X, we assign $S(0)$ to $\mathsf{2D\text{-}FindMin}(\mathsf{v}(0)+\mathsf{f}(0), L)$. If $S(0) = x \neq \#$, then we apply the operation $\mathsf{Delete}(x)$ followed by the operation $\mathsf{Insert}(T)$ where

$$T = (x, \mathsf{VirtualReleaseSlot}(S, x, 1, L), \mathsf{VirtualDeadline}(S, x, 1, L)).$$

Assuming that we have computed $S(0)$ through $S(t-1)$ for some $t > 0$, we now determine $S(t)$ as follows. First, we compute $\mathsf{v}(t) = \mathsf{v}(t-1) + \mathsf{f}(t-1)$ and the scaling factor $\mathsf{f}(t)$ (given $\mathsf{f}(t-1)$, this is easily accomplished at a cost that is linear in $|\Gamma_{t-1} \setminus \Gamma_t| + |\Gamma_t \setminus \Gamma_{t-1}|$). For each task x in $\Gamma_{t-1} \setminus \Gamma_t$, we perform a $\mathsf{Delete}(x)$ operation. For each task x in $\Gamma_t \setminus \Gamma_{t-1}$, we perform the operation $\mathsf{Insert}(T)$ where

$$T = (x, \mathsf{VirtualReleaseSlot}(S, x, t, L), \mathsf{VirtualDeadline}(S, x, t, L)).$$

Having properly updated the set X, we assign $S(t)$ to $\mathsf{2D\text{-}FindMin}(\mathsf{v}(t) + \mathsf{f}(t), L)$. If $S(t) = x \neq \#$, then we apply the operation $\mathsf{Delete}(x)$ followed by the operation $\mathsf{Insert}(T)$ where

$$T = (x, \mathsf{VirtualReleaseSlot}(S, x, t+1, L), \mathsf{VirtualDeadline}(S, x, t+1, L)).$$

We remark that if T' is the triple returned by $\mathsf{Delete}(x)$, then $T = (x, T.a + (1/\mathsf{r}(x)), T.b + (1/\mathsf{r}(x)))$. Thus it is easy to calculate T in constant time. (The same remark holds for the case $t = 0$ discussed earlier.)

The correctness of the above implementation of Algorithm BC follows immediately from Lemma 4 and the observation that the virtual release time or virtual deadline of a task x only needs to be updated when x is scheduled. The performance bounds claimed in Theorem 3 are straightforward to verify.

6 Concluding Remarks

We have addressed the problem of sharing a resource among a set of contending tasks, where: (i) only one task is allowed to use the resource at a time, (ii) the resource is scheduled in unit-time intervals, (iii) each task requires a specific fraction of the resource capacity over an extended period, and (iv) tasks arrive and depart at any time. We provided a formal criterion, P-fairness, to evaluate the fairness of such systems, and presented an efficient on-line P-fair scheduling algorithm.

In related work [1,3], efficient P-fair algorithms have been developed for the *multiple-resource periodic scheduling problem*, which may be viewed as the multiple-resource version of Problem A: (i) there is a static set of tasks, (ii) each task has constant weight less than or equal to 1, (iii) there are $m \geq 1$ resources, (iv) the sum of the task weights is at most m, and (v) up to m *distinct* tasks can be scheduled in each slot. Interestingly, the known P-fair scheduling algorithms for the multiple-resource periodic scheduling problem are quite a bit more complicated than Algorithm A. (The obvious generalization of Algorithm A to the case of multiple-resources is known not to be P-fair.) Given Theorem 3, an interesting open question is whether a polynomial-time P-fair on-line scheduling algorithm exists for the multiple-resource version of either Problem B or C.

Finally, the reader may wonder why we have not presented an on-line algorithm for Problem D (or E). The reason is that, in the present paper, we have focused our attention on the development of P-fair scheduling algorithms (note that all of the scheduling algorithms discussed in this paper, both off-line and on-line, are P-fair) and, unfortunately, Problems D and E do not admit such P-fair scheduling algorithms in the on-line case. To prove this claim, consider the following instance of Problem D (or E). At each slot t, the request $\mathsf{r}(x,t)$ of a task x is determined by the set of tasks

$$X_t = \bigcup_{\substack{0 \leq i < t \\ S(i) \neq \#}} S(i)$$

as follows: $\mathsf{r}(x,t) = 0$ if x belongs to X_t and $\mathsf{r}(x,t) = 1$ otherwise. Let $n = |\Gamma|$ and let t' denote the least slot t such that $|X_t| = n$. (If t' is not well-defined then some task is never assigned the resource and the lag of that task goes to infinity.) Let $x = S(t'-1)$ and observe that

$$\begin{aligned} \mathsf{Lag}(S, x, t') &\geq H_n - 1 \\ &= \Theta(\log n), \end{aligned}$$

where $H_n = \sum_{1 \leq i \leq n} 1/i$ denotes the nth harmonic number. This bound demonstrates not only that there is no on-line P-fair scheduling algorithm for Problem D (or E), but also that there is no on-line scheduling algorithm for Problem D (or E) that achieves *any* constant lag bound.

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, J. E. 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.

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

[5] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[6] R. L. Cruz. A calculus for network delay, Part I: Network elements in isolation. *IEEE Transactions on Information Theory*, 37:114–131, January 1991.

[7] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *Journal of Internetworking Research and Experience*, pages 3–26, September 1990.

[8] U. Maheshwari. Charge-based proportional scheduling. Technical Memorandum, MIT/LCS/TM–529, Laboratory for Computer Science, Massachusetts Institute of Technology, July 1995.

[9] E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14:257–276, 1985.

[10] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, June 1993.

[11] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the multiple node case. *IEEE/ACM Transactions on Networking*, 2(2):137–150, April 1994.

[12] I. Stoica and H. Abdel-Wahab. Earliest eligible virtual deadline first: A flexible and accurate mechanism for proportional share resource allocation. Technical Report TR–95–22, Department of Computer Science, Old Dominion University, November 1995.

[13] I. Stoica and H. Abdel-Wahab. A new approach to implement proportional share resource allocation. Technical Report TR–95–05, Department of Computer Science, Old Dominion University, April 1995.

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

[15] R. Tijdeman. The chairman assignment problem. *Discrete Mathematics*, 32:323–330, 1980.

[16] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1995.

[17] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating System Design and Implementation*, pages 1–12, November 1994.

[18] C. A. Waldspurger and W. E. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical Memorandum, MIT/LCS/TM–528, Laboratory for Computer Science, Massachusetts Institute of Technology, July 1995.