Minimizing Weighted Flow Time

N. Bansal * K. Dhamdhere †

October 6, 2002

Abstract

We consider the problem of minimizing *weighted* flow time on a single machine in the preemptive setting. Our main result is an $O(\log W)$ competitive online algorithm where the maximum to the minimum ratio of weights is W. More generally our algorithm achieves a competitive ratio of k if there are k weight classes. This gives the first $O(1)$-competitive algorithm for constant k. No $O(1)$ competitive algorithm was known previously even for the special case of $k = 2$. These results settle a question posed by Chekuri et al [5] about the existence of a "truly" online algorithm with a non-trivial competitive ratio. We also give a "semi-online" algorithm with competitive ratio $O(\log n + \log P)$, where P is ratio of the maximum to minimum job size. Our second result deals with the non-clairvoyant setting where the job sizes are unknown (but the weight of the jobs are known). We consider the resource augmentation model, and give a non-clairvoyant online algorithm, which if allowed a $(1 + \epsilon)$ speed-up, is $(1+1/\epsilon)$ competitive against an optimal offline, clairvoyant algorithm.

1 Introduction

We consider the classic problem of scheduling a collection of dynamically arriving jobs over time so as to minimize the total weighted flow time. The flow time of a job (also known as the response time) is the total time it spends in the system, thus it is the sum of times the job is waiting and its processing time. In the case when jobs have different degrees of importance, indicated by the weight of the job, the total (average) weighted flow time is one of the simplest and natural metrics that measures the quality of service received by the jobs.

For the unweighted case, a well known result [13] is that the online algorithm that at any time schedules the job with the shortest remaining processing time (SRPT) minimizes the flow time on a single machine. The weighted case however, was shown to be NP-hard by Lenstra et al [10]. The strong NP-hardness of the weighted flow time problem follows directly from the hardness of the related problem of minimizing weighted completion time. Note that the completion time differs from the flow time in that the flow time of a job is the difference of its completion time and its arrival time. In the last few years the weighted completion problem received significant attention and culminated in polynomial time approximation schemes for various versions of the problem [7, 1, 6]. However, the flow time metric is much more sensitive to small changes in schedule as compared with the completion time metric. Hence, none of the algorithmic techniques for weighted completion time give good results for weighted flow time.

Recently, there has been a lot of interested in weighted flow time, initiated mostly due to the first break through results of Chekuri et al [5, 4]. In [5], Chekuri, Khanna and Zhu give an $O(\log^2 P)$ competitive semi-online[1] algorithm , where P is the ratio of the maximum and minimum processing times of the jobs. They also give a lower bound of 1.618 on the competitiveness of any online algorithm. In the context of approximation, Chekuri and Khanna [4] gave a $1 + \epsilon$ approximation algorithm which has running time $O(n^{O(\ln W \ln P/\epsilon^3)})$, where W is the ratio of the maximum to the minimum job weight. Thus, their result gives a quasi-polynomial time approximation scheme (QPTAS) when both W and P are polynomially bounded in n. Moreover, they also give a QPTAS for the case when only one of either W or P is polynomially bounded in n. Surprisingly however, no polynomial time algorithm with $o(n)$ approximation ratio is known for the case of arbitrary W and P.

It is widely believed that the weighted flow time problem has a "truly" online (i.e. it is not semi-online) algorithm that is O(1) competitive. We make some progress towards this goal. Our main result is a truly online algorithm (which we call Balanced SRPT) that

*School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. nikhil@cs.cmu.edu Supported by IBM Research Fellowship.

†School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. kedar@cs.cmu.edu This research was supported in part by NSF ITR grants CCR-0085982 and CCR-0122581.

[1]It is semi-online in the sense that it uses the knowledge of P while scheduling the jobs.

is k competitive if the job weights belong to at most k different arbitrary weight classes. As a corollary, this yields an $O(1)$-competitive algorithm if $k = O(1)$. We wish to point out that even in the case when $k = 2$, the algorithm in [5] is $\Omega(\log P)$ competitive. Also, observe that Balanced SRPT directly gives a $O(\log W)$ competitive algorithm, since we can simply round the weights up to a power of 2. This rounding can affect the solution by a factor of at most 2, and since we have $\log W$ classes, apply Balanced SRPT to the rounded instance yields a $2 \log W$ competitive algorithm.

While the bounds of our algorithm are in terms of W, we also show that Balanced SRPT can be modified to give a semi-online algorithm with competitive ratio $O(\log n + \log P)$. Our algorithm is semi-online in the sense that it assume the knowledge of n,P and W while scheduling the jobs. In particular, our semi-online algorithm performs better than that in [5], unless $P = 2^{o(\sqrt{\log n})}$.

Our second result deals with the non-clairvoyant setting where the job sizes are unknown throughout its execution, i.e. the size of a job becomes known only when the job finishes its service requirement and leaves the system. The non-clairvoyant setting, introduced by Motwani, Phillips and Torng [11], realistically models many situations. For example. in the case of Unix operating system, it is usually not clear what is the CPU requirement of a job when it arrives. As can be expected, the performance of non-clairvoyant algorithms is quite bad as compared with the optimum clairvoyant and offline adversary. Indeed, Motwani *et al.* [11] show that any non-clairvoyant deterministic (resp. randomized) algorithm is at least $\Omega(n^{1/3})$ (resp. $\Omega(\log n)$) competitive for unweighted flow time. To overcome the problem of overly pessimistic bounds as given by the traditional competitive analysis, Kalyanasundaram and Pruhs [9] introduced the paradigm of *Resource Augmentation*. The idea is to augment the resources of the online scheduler by increasing its speed. Resource augmentation is widely used to obtain results in both clairvoyant and non-clairvoyant settings. Related to our work, [9] consider the non-clairvoyant minimization of the total unweighted flow time. They give an online algorithm which is $(1 + 1/\epsilon)$ competitive, when provided with a $(1 + \epsilon)$ times faster processor than the optimum offline clairvoyant algorithm. For the problem of weighted flow time, Becchetti *et al.* [2] recently gave an algorithm that is $(1 + \frac{1}{\epsilon})$-speed, $(1 + \epsilon)$-competitive. However, their algorithm works in a *clairvoyant* setting.

In this paper, we give a $(1 + \epsilon)$-speed $(1 + \frac{1}{\epsilon})$-competitive algorithm for minimizing the total weighted flow time in the *non-clairvoyant* setting. Thus our result can be viewed as an extension of the Kalyanasundaram–Pruhs [9] result for the weighted case, as well as an extension of Becchetti *et al.* [2] for the non-clairvoyant case. What is interesting is that our algorithm matches the performance of the algorithm of Becchetti *et al.* [2], while being *non-clairvoyant*.

Model: We are given a set J of n jobs that are released over time. For a job $x \in J$, the quantities $r(x)$, $|x|$ and $w(x)$ denote the release time, processing time (or size) and the weight respectively. The flow time of x is $F(x) = C(x) - r(x)$, where $C(x)$ is the completion time of job x in the schedule. Our objective is to minimize $\sum_{x \in J} w(x)F(x)$, the total weighted flow time. We consider a single machine setting where job preemptions are allowed. In the clairvoyant setting we assume that processing time of a job is known upon its arrival, while in the non-clairvoyant setting we assume that the processing time is unknown throughout the time the job is present in the system. In either setting, the weight of a job is known when it arrives.

Proof Techniques: We classify the jobs into different classes according to their weights. With a weight class our algorithm follows SRPT. At any moment of time the decision of which class to work on is made depending on the weights of the jobs in each weight class. This suggests a general paradigm where one can achieve an $O(k)$ competitive algorithm for k weights classes using an optimal algorithm for the unweighted case within each weight class. While this paradigm is fairly natural, the argument is usually non-trivial and not many results are known that use it. The only result we are familiar with is for weighted caching, due to Irani [8]. They use the randomized marking algorithm within each weight class and decide which page to evict based on a global criteria. Their analysis is quite involved, and currently it is not clear how to extend their analysis beyond the case where there are more than two types of weights.

Our second result for non-clairvoyant weighted flow time combines the techniques of [2] and [9]. In [2], the idea is to work on the jobs with high density (defined as weight of a job divided by its size). Since the sizes are not known in our setting, we use the ideas similar to [9] to guess the size.

2 Preliminaries

To get a feel for the problem, we first describe some simple algorithms and give reasons for their poor performance. Secondly, we show by an example that the algorithm of Chekuri *et al.* [5] has a competitive ratio of $\Omega(\log P)$ even in the case when the job weights are either 1 or 2. We begin by stating the notion of local-competitiveness, which will be used throughout the paper.

Local Competitiveness: Let O be the optimum algorithm and A some other algorithm. Let $Wg_o(t)$ and $Wg_a(t)$ denote the total weight of jobs present in O's and A's system respectively at time t. The total weighted flow time under A is given by $\int_0^\infty Wg_a(t)dt$. Notice that if $Wg_a(t) \leq cWg_o(t)$ for some c at all times t, then A is c-competitive. In this case we say that the A is *locally* $c-competitive$. While local c-competitiveness is sufficient for global c-competitiveness, it also turns out to be necessary [2]. The idea is that if $Wg_a(t) > cWg_o(t)$ at some time t, then the adversary can start giving a stream jobs of size $\epsilon \to 0$ and weight 1 every ϵ units of time. This forces both O and A to work on the new stream. By making the stream long enough, adversary can force the competitive ratio to be greater than c.

Thus it is essential to keep the total weight of the jobs in our algorithm low at all times. A natural strategy to consider is greedy, which at any time instant schedules the job with the highest weight to remaining time ratio. However this is easily shown to be $\Omega(\sqrt{P})$ competitive [5]. The idea is that breaking ties badly, the naive greedy algorithm may be left with a large job while the optimum does not have this difficulty. Similarly, the algorithm which simply disregards the size of the job and works on the highest weight job is easily shown to be $P-competitive$ [5]. The $O(\log^2 P)$ competitive algorithm of Chekuri *et al.* deals with this problem by striking a balance. It processes a large weight job without regard to its ratio as long as the total weight of jobs with a strictly better ratio does not get too large. However, even when the job weights are just 1 or 2, it can be shown that their algorithm is $\Omega(\log P)$ competitive[2]. For completeness, their algorithm is described in Appendix A. We now describe our example.

Consider the following scenario: At time $t = 0$ two jobs, one of size P and weight 2 and other of size $P/2$ and weight 1 are released. For i ranging from 1 to $\log P - 1$, at each time instant $P(1-2^{-i})$ a job of weight 1 and size $P/2^{i+1}$ is released. Finally, at time $t = P(1-2^{i+1})$, another job of size $P/2^{i+1}$ is released.

At time $t = 0$, their algorithm will schedule the job of weight 2 and continue executing till time P. At this stage the weight of the jobs in the $Q(t)$ will be $(\log P + 1)$. The optimum algorithm on the other hand works only on jobs of weight 1 and hence finishes all jobs except the one of weight 2 by time P. Thus the algorithm is locally $\frac{1}{2}(\log P + 1)$ competitive, and hence $\Omega(\log P)$ competitive.

The reason for the bad performance of the algorithm above is that it continues to work on the high weight job even when a lot of smaller weight jobs keep accumulating. Our algorithm will be careful in this respect and will in fact be 2-competitive for the 2 weight case. We next describe our algorithm and analyze it.

3 Algorithm

We now present an k-competitive algorithm, that we call *Balanced SRPT* for minimizing weighted flow time. We then analyze it and show that its analysis is tight. Finally we show how this also gives a $\log n + \log P$ competitive semi-online algorithm.

We assume that the weights of the jobs are drawn from the set $\{a_1, a_2, \ldots, a_k\}$, where $a_1 < a_2 \ldots < a_k$. We will refer to these as weight classes $1, \ldots, k$. Without loss of generality[3], we also assume that a_i/a_{i-1} is integral for $1 < i \leq k$. Let X be a set of jobs. We use $Wg(X)$ to mean the sum of the weights of the jobs in X. We partition the sets of jobs that are alive at a specific time based on the weight class. Let $Q_j(t)$ (or just Q_j when the time t is evident from the context) denote the set of alive jobs of class j at time t.

Balanced SRPT: At all times, pick a Q_j which has maximum $Wg(Q_j(t))$. (Break ties in favor of higher weight class). Execute the job with shortest remaining processing time from Q_j.

Informally speaking, the algorithm tries to balance the weight in various weight classes, while giving priority to larger weights. It runs SRPT within each weight class. Note in particular that the algorithm favors jobs with a higher weight. For example, if there are $W-1$ (or even W) jobs of weight 1 and one job of weight W. The algorithm works on the W weight job first.

3.1 A Simple Example

While our algorithm is extremely simple to describe, its analysis is quite involved. The following example illustrates why. We will show that it could be the case that Balanced SRPT has more jobs in *each* weight class as compared with the optimum algorithm.

Suppose are time $t = 0$, we get 2 jobs of size 1 and weight 1, and 1 job of size 3 and weight 2. At time $t = 2$, we get 3 jobs of size ϵ and weight 1, and 1 job of size ϵ and weight 2.

The Balanced SRPT algorithm would work on the job of size 3 and weight 2 for the first two time units,

[2] Their algorithm rounds up the weights of incoming jobs to an integral power of $4(\log P + 1)$. Hence, it is trivially loses a $4(\log P + 1)$ factor in competitive ratio. However, our example does not use this rounding to prove $\Omega(\log P)$ competitiveness.

[3] Arbitrary weights can be rounded up to an integral power of 2 and we get the required property of integral ratios, while losing only a factor of 2 in competitive ratio.

then it would switch to jobs of weight 1 as they have more weight. Note that at time $2+\epsilon$ the queue contains 4 jobs of weight 1 and 2 jobs of weight 2. The optimum on the other hand can work on the jobs of weight 1 for the first two time units, and then switch to a job of weight 2. Note that at time $2+\epsilon$, there are 3 jobs of weight 1 left and 2 jobs of weight 2. Thus, each weight class has more number of jobs (hence weight) under Balanced SRPT. In particular, note that even though Balanced SRPT has spent more time than the optimum on jobs of weight 2, it can contain more jobs of weight 2 than the optimum.

Intuitively, this happens because Balanced SRPT might be working on the wrong jobs at the wrong time. The idea of our analysis will be an extension of an idea used by Schrage in his proof that at any time SRPT has the minimum number of jobs present in the system [12]. Roughly our argument will be the following: We will form prefixes of jobs under Balanced SRPT in a way such that, if we consider a prefix of jobs with total weight W' and total remaining processing time P', then any collection of jobs under Opt with remaining processing time at least P' will have weight at least W'/k. Now, since both Balanced SRPT and Opt are work-conserving algorithms (i.e. work on a job whenever there is a job to work on), both will have an identical amount of remaining processing times at all times. This will imply the k competitiveness of Balanced SRPT.

3.2 Analysis We now prove the k competitiveness of the Balanced SRPT algorithm and also show that the analysis is tight.

First we give some notation for describing some useful quantities. We denote our Balanced SRPT algorithm by B and the optimum algorithm by Opt. The tuple (j,l) denotes job in weight class j which has the l^{th} largest remaining processing time among jobs in that class. Define an order on 2-tuples of integers: $(j,l) \prec (j',l')$ iff $a_j \cdot l \leq a_{j'} \cdot l'$ for $j < j'$ or $a_j \cdot l < a_{j'} \cdot l'$ for $j \geq j'$. Also, we say $(j',l') \preceq (j,l)$ if $(j,l) \not\prec (j',l')$. Note that $(j',l') \preceq (j,l)$ and $(j,l) \preceq (j',l')$ holds iff $j=j'$ and $l=l'$, and that $\preceq$ defines a total order on the 2-tuples.

Intuitively, the notion $\prec$ formalizes the order in which our algorithm B executes the jobs. If there were l jobs of class j and l' jobs of class j', then $(j,l) \prec (j',l')$ means that algorithm B will execute a job of class j' first (assuming no new jobs arrive).

Let $Q_B(t)$ denote the set of jobs that aren't finished by algorithm B by time t. We define a set of jobs $B(j,l,t)$ as follows:

$B(j,l,t) = \{x \in Q_B(t) \mid \text{class}(x) = j' \ \& \ x \text{ has } l'\text{th largest remaining time in class } j' \ \& \ (j',l') \preceq (j,l)\}$

Let $|B(j,l,t)| = \sum_{x \in B(j,l,t)} rem(x,t)$, where $rem(x,t)$ is the remaining time of job x at time t.

The main proof idea will be to compare the "prefixes" $|B(j,l,t)|$ with suitable prefixes of Opt. Note that $|B(j,l,t)|$ denotes the total remaining processing in a subset of jobs defined by the total order $\preceq$. Next, we will define an order π on the jobs in the queue of Opt. This order will have the special property that if we consider a subset of jobs S in Opt which is chosen according to the order π, then a corresponding subset of jobs of type $B(j,l,t)$ in B will be produced which has more total remaining time than in S, and whose total weight is not more than k times the weight of S. We will finally show how this proves the k competitiveness.

We now define the order π on jobs of Opt. Let $Q_{Opt}(t)$ denote the set of jobs that are not finished by Opt by time t. Along with these jobs, we also want to include some *"dummy"* jobs in $Q_{Opt}(t)$. A *dummy* job of class j, is a job with remaining time 0. Let $\hat{Q}_{Opt}(t)$ denote the set $Q_{Opt}(t)$ along with some dummy jobs thrown in. These dummy jobs play a role only in simplifying the proofs.

Consider a permutation π of jobs in $\hat{Q}_{Opt}(t)$. Call π a *valid ordering* of $\hat{Q}_{Opt}(t)$ if jobs i & j belong to same class then $\pi(i) < \pi(j) \Rightarrow rem(\pi(i)) \geq rem(\pi(j))$. We don't impose this condition if the jobs i and j don't belong to same class. Intuitively, all the jobs from same class in $\hat{Q}_{Opt}(t)$ appear in π in descending order of their remaining times. Note, that there are many possible choices of π. In particular, the jobs of different classes in $Q_{Opt}(t)$ could be interleaved arbitrarily. We will see later that this flexibility in the choice of π plays a very crucial role in the proofs.

For a job $x \in \hat{Q}_{Opt}(t)$ and a valid ordering π of $\hat{Q}_{Opt}(t)$, we define $Opt_\pi(x,t) = \{y \in \hat{Q}_{Opt}(t) \mid \pi(y) \leq \pi(x)\}$. In other words, $Opt_\pi(x,t)$ contains the jobs that precede x in the valid ordering π. We call $Opt_\pi(x,t)$ a "prefix" of Opt under the valid ordering π. If j is the class of job x and $l = \lceil Wg(Opt_\pi(x,t))/w(x) \rceil$, then we can think of $Opt_\pi(x,t)$ as consisting of l jobs of class j and thus write $Opt_\pi(j,l,t)$ as a synonym for $Opt_\pi(x,t)$. Let $|Opt_\pi(j,l,t)|$ denote the sum of remaining times of the jobs in $Opt_\pi(j,l,t)$. For a job x, we will compare $|Opt_\pi(x,t)|$ (which is same as $|Opt_\pi(j,l,t)|$ with the prefix $|B(j,l,t)|$ of algorithm B. $Opt_\pi(j,l,t)$ has weight about k times smaller than $B(j,l,t)$. It's important to note here that, $Opt_\pi(j,l,t)$ may not be defined for all pairs (j,l) unlike that for $B(j,l,t)$. Specifically, $Opt_\pi(j,l,t)$ is defined only if $\exists y \in \hat{Q}_{Opt}(t)$ such that $Opt_\pi(y,t) = Opt_\pi(j,l,t)$.

Finally, let $Rem(B,t) = \sum_{x \in Q_B(t)} rem(x,t)$ and $Rem(Opt,t) = \sum_{x \in Q_{Opt}(t)} rem(x,t)$. [Note that:

$Rem(B,t) = Rem(Opt,t)]$

With this notation in place, we can now state the "domination" property of prefixes as follows.

LEMMA 3.1. *At all times t, for any valid ordering π of $\hat{Q}_{Opt}(t)$, for $1 \le j \le k$ and $l \ge 1$, the following holds*

$$|B(j,l,t)| \ge |Opt_\pi(j,l,t)|$$

whenever the latter is defined.

Proof. We prove the result by induction over time. We know that the invariant holds when there are no jobs present just before time $t = 0$. Now we need to verify that, it still continues to hold at various events. We analyze following possible cases:

(1) **Neither arrival nor departure** If in an interval $[t_1, t_2)$ there are no arrivals and neither algorithm B nor Opt finishes any job. Let x be the job on which B is working. We observe that $|B(j,l,t_2)| < |B(j,l,t_1)|$ iff $x \in B(j,l,t_1)$. For $t \in [t_1, t_2)$, $x \in B(j,l,t)$ means that $\forall x' \in Q_B(t), x' \preceq x \Rightarrow x' \in B(j,l,t)$. Thus $|B(j,l,t)| = Rem(B,t)$. Thus, we can say that, $|B(j,l,t_2)| = \min(|B(j,l,t_1)|, Rem(B,t_2))$.

For a fixed ordering π of $\hat{Q}_{Opt}(t_1)$ and a job $y \in \hat{Q}_{Opt}(t_1)$ (s.t. $Opt_\pi(y,t_2) = Opt_\pi(j,l,t_2)$), we have $|Opt_\pi(y,t_2)| \le |Opt_\pi(y,t_1)|$. Moreover, $|Opt_\pi(y,t_2)| \le Rem(Opt,t_2)$. Therefore, we have $|Opt_\pi(j,l,t_2)| \le \min(|Opt_\pi(j,l,t_1)|\ , Rem(Opt,t_2))$.

Putting the two together, it follows that $|B(j,l,t)| \ge |Opt_\pi(j,l,t)|$, for $t \in [t_1,t_2)$.

(2) **Departure** At time t, algorithm B finishes a job. Let $t' < t$. Observe that, $\forall x \in Q_B(t'), rem(x,t')$ is left continuous. Hence $|B(j,l,t')|$ is left continuous. If Opt hasn't finished a job at t, $|Opt_\pi(j,l,t)|$ is clearly left continuous. The inequality then follows.

If at time t, Opt finishes a job y. Let $t' < t$ and consider the limit $t' \to t$. Consider $\hat{Q}_{Opt}(t') = \hat{Q}_{Opt}(t) \bigcup \{y\} - \{$a dummy job of same class that of $y\}$. If there is no dummy job with same class as that of y, then $\hat{Q}_{Opt}(t') = \hat{Q}_{Opt}(t) \bigcup \{y\}$. Let π' be the ordering of $\hat{Q}_{Opt}(t')$ obtained by appending y at the end of π. (If there were any dummy jobs in π in the same class as that of y, then y takes the place of the first such dummy job). Since $rem(y',t')$ is left continuous for every $y' \in \hat{Q}_{Opt}(t)$. Therefore, we have $|Opt_{\pi'}(y',t')| = |Opt_\pi(y',t)|$, in limit. (If there was a dummy job in $\hat{Q}_{Opt}(t) - \hat{Q}_{Opt}(t')$, then we also have $|Opt_{\pi'}(y,t')| = |Opt_\pi(\text{dummy},t)|$ in limit). This proves the left-continuity of all $|Opt_\pi(j,l,t)|$. If algorithm B hasn't finished a job at time t. $|B(j,l,t)|$ is also clearly left continuous since each of $rem(x,t)$ is left continuous for $x \in B(j,l,t)$.

(3) **Arrival** The most troublesome case is the arrival of a new job. In this case, neither $|B(j,l,t)|$ nor $|Opt_\pi(j,l,t)|$ will be left-continuous, in general.

Let x denote the newly arrived job. Let $|x|$ denote its size. And let $c = class(x)$. Let $t' < t$. Let's look at the limit $t' \to t$. So that by $|B(j,l,t')|$ etc, we mean the left limits. Since we arrange the jobs in decreasing order of remaining times within a class, we have the following:

FACT 3.1. *If $x \in B(j,l,t)$, then $|B(j,l,t)| \ge |B(j,l,t')|$. And if $x \notin B(j,l,t)$, then $|B(j,l,t)| = |B(j,l,t')|$.*

Using this fact, we can handle the case when $x \notin Opt_\pi(j,l,t)$. Consider $\hat{Q}_{Opt}(t') = \hat{Q}_{Opt}(t) \bigcup \{$a dummy job of class $c\} - \{x\}$. And let π' be an ordering on $\hat{Q}_{Opt}(t')$ s.t. $\forall i \ge 1$, $class(\pi(i')) = class(\pi(i))$. [4] Then, $x \notin Opt_\pi(j,l,t)$ means that $|Opt_\pi(j,l,t)| = |Opt_{\pi'}(j,l,t')|$. And hence using the inequality at time t' for the ordering π', we get $|B(j,l,t)| \ge |Opt_\pi(j,l,t)|$.

Therefore, we only need to consider the case when $x \in Opt_\pi(j,l,t)$. We split this in three different cases ($j > c$, $j = c$ and $j < c$). For each of these sub-cases, we use the inequality at time t' with a different ordering on $\hat{Q}_{Opt}(t')$.

FACT 3.2. *If $B(j,l,t)$ contains atleast one more job x' of class $c = class(x)$ than $B(j',l',t')$, then $|B(j,l,t)| \ge |B(j',l',t')| + |x|$.*

Proof. If $x \notin B(j,l,t)$, then $rem(x',t) \ge |x|$, thus giving us required result. Otherwise if $x \in B(j,l,t)$, then $B(j,l,t') = B(j,l,t') \bigcup \{x\} - \{x''\}$ for some job x'' s.t. $class(x'') = class(x)$ and $rem(x'')$ is the smallest among all the jobs of same class in $B(j,l,t')$. Thus $rem(x',t) \ge rem(x'',t)$ and the result follows.

case (a) $j = c$. Let y be the job such that $Opt_\pi(y) = Opt_\pi(j,l,t)$. First, consider a different ordering π' of $\hat{Q}_(t)$, which reorders the jobs that precede y in the ordering π, so that in π', they are are in decreasing order of weight class (and within each class, in decreasing order of remaining times). Let z be the job in π' just preceding y. Let $Opt_{\pi'}(z,t) = Opt_{\pi'}(j',l',t)$. Then, $Wg(Opt_{\pi'}(z,t)) = Wg(Opt_{\pi'}(y,t)) - a_j$.

Now, if $j' \le j$, it's easy to see that $(j',l') \preceq (j,l-1) \prec (j,l)$. On the other hand, if $j' > j$, then all the jobs preceding y have weight integral multiple

[4] Note that, specifying class of jobs at each place in π' defines it uniquely, as we require the jobs within each class to be in descending order of remaining time.

of a_j. Hence $Wg(Opt_{\pi'}(z,t))$ is an integral multiple of a_j. Thus it follows that, $(j',l') \prec (j,l)$. It means that $B(j,l,t)$ contains atleast one more job of class j than $B(j',l',t')$. And $|B(j,l,t)| \geq |B(j',l',t')| + |x|$ follows from fact 3.2.

With this knowledge, we are now ready to use the inequality at time t'. Let $\hat{Q}_{Opt}(t') = \hat{Q}_{Opt}(t) \bigcup \{\text{a dummy job of class } c\} - \{x\}$. And let σ be an ordering of $\hat{Q}_{Opt}(t')$ s.t. $\forall i \geq 1$, $class(\sigma(i)) = class(\pi'(i))$. It's easy to see that $|Opt_{\pi}(y,t) = |Opt_{\pi'}(y,t)| = |Opt_{\sigma}(j',l',t')| + |x| \leq |B(j',l',t')| + |x| \leq |B(j,l,t)|$.

case (b) $j < c$. Let y be the job such that $Opt_{\pi}(y,t) = Opt_{\pi}(j,l,t)$. We also have $x \in Opt_{\pi}(y,t)$. x is not same as y. Now consider $\hat{Q}_{Opt}(t') = \hat{Q}_{Opt}(t) - \{x\}$. Let σ be an ordering of $\hat{Q}_{Opt}(t')$ obtained by just deleting x from π. Therefore, we have $|Opt_{\sigma}(y,t') + |x| = |Opt_{\pi}(y,t)|$. Let $Opt_{\sigma}(y,t') = Opt_{\sigma}(j,l',t')$. Then, we have $(j,l') \prec (c, \lfloor (a_j l)/a_c \rfloor) \prec (j,l)$. This means that, $B(j,l,t)$ contains atleast one extra job of weight c than $B(j,l',t)$. By applying, fact 3.2 we get the desired result.

case (c) $j > c$. Let y be the job such that $Opt_{\pi}(y,t) = Opt_{\pi}(j,l,t)$. Consider an ordering π' of $\hat{Q}_{Opt}(t)$ which reorders the jobs in $Opt_{\pi}(y,t)$ s.t. jobs of class c in $Opt_{\pi}(y,t)$ are all preceded by all other jobs. π' retains the order of the jobs not in $Opt_{\pi}(y,t)$ same as in π. Let z be the job of class c that ends up last among the jobs from $Opt_{\pi}(y,t)$ in ordering π'.

Clearly, $|Opt_{\pi}(y,t)| = |Opt_{\pi'}(z,t)| = |Opt_{\pi'}(c,l',t)|$ for some l'. From case (a), we get $|B(c,l',t)| \geq |Opt_{\pi'}(c,l',t)|$. And we can easily prove that $(c,l') \prec (j,l)$, whereby we get $|B(j,l,t)| \geq |B(c,l',t)| \geq Opt_{\pi}(j,l,t)|$.

Thus in all possible cases we have shown that the inequality holds.

THEOREM 3.1. *Balanced SRPT is a k-competitive online algorithm for minimizing total weighted flow time.*

Proof. We will prove that B is locally k-competitive, which gives us the desired result.

Fix a time t. Suppose algorithm B was working on a job x at the time t. Let $j = class(x)$. For $1 \leq i \leq k$, let n_i denote the number of jobs of class i in $Q_B(t)$. In particular, x is n_jth job of class j. For $\neq j$, we know that $(i,n_i) \prec (j,n_j)$. Thus $a_j \cdot n_j \geq a_i \cdot n_i$ for all i s.t. $1 \leq i \leq k$. Hence we get,

$$Wg(B) \leq k \cdot n_j \cdot a_j \tag{3.1}$$

Now we derive a lower bound on $Wg(Opt)$. Consider an ordering π of $Q_{Opt}(t)$ in which jobs are ordering from the highest weight class to the lowest one. Note that we don't include any *dummy* jobs here. In particular, if y is the last job in π, then y has the lowest weight class (and smallest remaining time among it's weight class). Let $i = class(y)$, and let $l = \lceil Wg(Opt)/a_i \rceil$. Note that since i is the smallest weight class in Opt, $Wg(Opt)$ is an integral multiple of a_i and hence $l = Wg(Opt)/a_i$.

We first note that $(i,l) \succeq (j,n_j)$. This follows since, if $(i,l) \prec (j,n_j)$ then $x \notin B(j,n_j,t)$ and hence $B(i,l,t) < B(j,n_j,t)$, thus $Rem(B,t) = B(j,n_j,t) > B(i,l,t) \geq Opt_{\pi}(i,l,t) = Rem(Opt,t)$ which violates $Rem(B,t) = Rem(Opt,t)$. The third inequality above follows from Lemma 3.1 and final equality follows from our choice of π.

We can now show that $Wg(Opt) \geq n_j \cdot a_j$. Suppose $Wg(Opt) < n_j \cdot a_j$, then $a_i l < a_j n_j$. For either case, when $i < j$ or $i \geq j$, the definition of the relation $\prec$ gives us that $(i,l) \prec (j,n_j)$. Thus we have,

$$\mathrm{W}g(Opt) \geq n_j \cdot a_j \tag{3.2}$$

From equations (3.1) and (3.2), local k-competitiveness of Balanced SRPT follows.

3.3 Analysis of Balanced SRPT is tight Finally, we show that the competitive ratio for the Balanced SRPT algorithm is tight. Consider following job instance, in which for $1 \leq j \leq k$, 2^{k-j} jobs of weight 2^j arrive at time 0. The job of weight 2^k has size 1 while rest of them have sizes $\epsilon = 2^{-k}$. Since all weight classes have an equal amount of weight, Balanced SRPT would start working on the job of weight 2^k and continue working on it till the job finishes.

Consider the scenario at time $t = (2^k - 2)\epsilon < 1$. At t, Balanced SRPT is still working of the job of weight 2^k, hence the total weight under Balanced SRPT is $k2^k$. Where as, the optimal algorithm can finish all the jobs of size ϵ first by time t. In this case, the optimal algorithm has a weight of 2^k. Thus Balanced SRPT is locally k competitive at time t, now we can make it globally k competitive, but giving a long stream of weight 2^k and size 2^{-k} jobs.

The example above also suggests that any O(1) competitive algorithm for the weighted flow time problem must be designed to carefully balance the number of jobs with high densities and the jobs with high weights.

3.4 The Semi-Online Algorithm Given an instance I, assume without loss of generality that the job sizes are integers and the smallest job size is 1. Suppose we know the values of the largest weight, largest job size and the total number of jobs (i.e. W, P and n). Then we can modify the Balanced SRPT algorithm in the following way: Round the weights up to the nearest power of 2. Consider only the jobs with weights between $\frac{W}{Pn^2}$

and W. Call these jobs *heavy*. Run the Balanced SRPT algorithm on the heavy jobs, if at some time no heavy job is present, work on any arbitrary job.

Let us the denote the instance when restricted to heavy jobs as I_h. Let $Opt(I_h)$ $A(I_h)$ denote the weighted flow time of the optimum algorithm and our algorithm on I_h respectively.

When restricted to heavy jobs, by Theorem 3.1, the performance of the Balanced SRPT is no worse than $4(\log n + \log P)$ times the optimum (restricted to I_h). Thus,

$$A(I_h) \leq 4(\log n+\log P)Opt(I_h) \leq 4(\log n+\log P)Opt(I)$$

Each non-heavy job can be delayed by at most by nP. Since its weight is no more than W/n^2P, it adds at most W/n to the total weighted flow time. Since there are at most n non-heavy jobs, the total contribution of these is no more than W. Since $Opt(I)$ is lower bounded by W, we get.

$$\begin{aligned} A(I) &\leq A(I_h) + W \\ &\leq 4(\log n + \log P)Opt(I) + W \\ &\leq 4(\log n + \log P)Opt(I) + Opt(I) \end{aligned}$$

Thus we get a semi-online algorithm which is $O(\log n + \log P)$ competitive.

4 Non-clairvoyant scheduling for weighted flow time

In this section, we consider the non-clairvoyant case where the processing time of a job becomes known only when it finishes. We now give a resource augmented $(1+\epsilon)$ speed $(1+1/\epsilon)$ competitive algorithm. We call it the *Weighted Foreground Background* (WFB) algorithm as it resembles the commonly used Foreground Background (FB) algorithm. Note that the competitive ratio is independent of the number of weight classes and also of the ratio of the jobs sizes. Moreover, it generalizes the results of [9] and [2] while still matching their bounds both in terms of resource augmentation and competitive ratio. In terms of proof techniques, our analysis builds up on the ideas in [9].

4.1 Algorithm Description For a job J_i, let $p_i(t)$ denote the amount of work done on that job by time t. And let w_i denote the weight of the job J_i. We define a norm of job J_i as $\|J_i\|_t = \frac{p_i(t)}{w_i}$. We say that a job J is alive at time t in some algorithm A if J has arrived but not yet finished its total processing requirement.

Algorithm **WFB**: At all times, WFB splits the processor, proportional to weights of the jobs, among the jobs J_i that have the smallest norm $\|J_i\|_t$. So, if $J_1, \ldots, J_k$ are the jobs with WFB that have the smallest norm. Then the job J_j will receive $w_j/(\sum_{i=1}^k w_i)$ fraction of the processor.

Note that, for all jobs J_i which WFB executes, the norm increases at the same rate and thus stays same. Moreover, if at some point of time t', we have $\|J_i\|_{t'} \geq \|J_j\|_{t'}$, then for all times t in future (i.e. $t \geq t'$), while both jobs are alive, we have $\|J_i\|_t \geq \|J_j\|_t$.

4.2 Analysis For an algorithm A with a speed s processor, let $Wg_A(t,s)$ denote the sum of weights of the jobs that are alive in A at time t. The analysis of WFB algorithm hinges on proving the following lemma:

LEMMA 4.1. *At any point t in time, we have the following relation:*

$$Wg_{WFB}(t, 1+\epsilon) \leq (1+\frac{1}{\epsilon})Wg_{Opt}(t,1)$$

In other words, $(1+\epsilon)$-speed WFB is locally $(1+1/\epsilon)$-competitive w.r.t. an adversary with a unit speed processor.

The lemma will be proved in argument that follows.

Let's fix a time t. Let V denote the set of jobs which are alive in WFB at time t but the adversary has finished by time t. And let U denote the set of jobs which the alive in the adversary's algorithm at time t. Let $Wg(V) = \sum_{i \in V} w_i$ and $Wg(U) = \sum_{j \in U} w_j$.

We want to bound $Wg(V)$ in terms of $Wg(U)$.

We say that a job J_i can *immediately borrow* from a job J_j ($J_i \hookleftarrow J_j$) if WFB ran J_j at some time while J_i was alive. We define the *borrow* relation as the transitive closure of immediately borrow. (We also denote this by $J_i \leftarrow J_j$)

LEMMA 4.2. *If $J_i \hookleftarrow J_j$, then we have $\|J_i\|_t \geq \|J_j\|_t$.*

Proof. It suffices to prove the assertion for immediately borrow relation. So assume that J_i immediately borrows from J_j. From the property of WFB, $\|J_i\|_{t'} \geq \|J_j\|_{t'}$ must have been true at some time t'. Thus from the observation we made above, the lemma follows.

We partition the set V into $V_1, V_2, \ldots, V_k$ such that $J_i, J_j \in V_{k'} \iff \|J_i\|_t = \|J_j\|_t$. We also require that, if $J_i \in V_{i'}$, $J_j \in V_{j'}$ and $i' < j'$ iff $\|J_i\|_t > \|J_j\|_t$. Let $\|V_i\|$ denote $\|J_i\|$, where $J_i \in V_i$.

Intuitively, all jobs with same norm are in the same partition. Also, V_1 has the jobs with largest norm, V_2 has jobs with second largest norm, and so on.

Let $U' = \{J_j \in U \mid \exists i \in V \ s.t. \ J_i \text{ borrows from } J_j\}$. These constitute the jobs among U, on which WFB worked while it had some jobs from V still alive.

We note that, WFB has a faster processor with speed $(1+\frac{1}{\epsilon})$. And while adversary finished all jobs in the set V by time t, WFB spent at least $\sum_{i\in V} \epsilon\cdot w_i\|J_i\|_t$ amount of work on jobs in U. In fact, the definition U' is such that, it contains all the jobs from U on which WFB worked when there was some job from V alive in WFB.

Let us define $U'_i = \{J_j \in U' \mid \|V_i\|_t \geq \|J_j\|_t > \|V_{i+1}\|_t\}$ (for notational convenience, assume that $\|V_{k+1}\|_t = 0$). By virtue of lemma 4.2, we know that $\|V_1\|_t \geq \|J_j\|_t$, for all jobs $J_j \in U'$. Thus, $\bigcup_i U'_i$ is indeed U'.

Moreover, note that for $J_i \in V_i$ and $J_j \in U'_j$, J_i borrows from J_j is possible only if $i \leq j$.

Let t_i be the time when a job from V_i arrived first. Note that $t_1 < t_2 < \ldots < t_k$. Consider an arrival time t_m. Since the time t_m, *Opt* has finished all jobs from V that arrived after time t_m. So the amount of work done by *Opt* is at least $\sum_{j=m}^{k}\sum_{i\in V_j} w_i\|J_i\|_t$. And the amount of work done by $(1+\epsilon)$-WFB is at least $\sum_{j=m}^{k}\sum_{i\in V_j}(1+\epsilon)w_i\|J_i\|_t$. WFB hasn't finished any of the jobs from V. So work done by WFB on the jobs from $\bigcup_{m\leq j\leq k} V_j$ is at most $\sum_{j=m}^{k}\sum_{i\in V_j} w_i\|J_i\|_t$. Thus WFB has worked at least $\sum_{j=m}^{k}\sum_{i\in V_j}\epsilon w_i\|J_i\|_t$ on jobs in $\bigcup_{m\leq j\leq k} U'_j$ after time t_m. Call this work T_m.

For $1 \leq m \leq k$ we have

$$T_m \geq \sum_{j=m}^{k}\sum_{i\in V_j}\epsilon w_i\|V_j\|_t \tag{4.3}$$

For a job $l \in U'_j$, work done on l, by time t is $\|J_l\|_t \leq \|V_j\|_t$, by the definition of U'_j. Thus summing up over jobs in $\bigcup_{m\leq j\leq k} U'_j$, we get

$$T_m \leq \sum_{j=m}^{k}\sum_{l\in U'_j} w_l\|V_j\|_t \tag{4.4}$$

We now use following inequality in conjunction with 4.3 and 4.4 to derive desired result.

LEMMA 4.3. *Let $y_1 \geq y_2 \geq \ldots \geq y_k > 0$. Let $x_i, z_i \geq 0$ for $1 \leq m \leq k$ and $\sum_{i=m}^{k} x_i y_i \geq \sum_{i=m}^{k} z_i y_i$. Then $\sum_{i=1}^{k} x_i \geq \sum_{i=1}^{k} z_i$.*

Proof. Let $X_m = \sum_{i=m}^{k} x_i y_i$ and $Z_m = \sum_{i=m}^{k} z_i y_i$. We have $X_m \geq Z_m$. For notational convenience let $y_0 = \infty$. Then,

$$\sum_{i=1}^{k} x_i = \sum_{i=1}^{k}\left(\frac{1}{y_i} - \frac{1}{y_{i-1}}\right)X_i \geq \sum_{i=1}^{k}\left(\frac{1}{y_i} - \frac{1}{y_{i-1}}\right)Z_i = \sum_{i=1}^{k} z_i$$

(Each $(1/y_i - 1/y_{i-1}) \geq 0$ since $y_i \leq y_{i-1}$ for $1 \leq i \leq k$.)

Now, we set $x_j = \sum_{i\in U'_j} w_i$, $z_j = \sum_{i\in V_j}\epsilon w_i$ and $y_j = \|V_j\|_t$. The lemma 4.3 tells us that

$$Wg(U) \geq \sum_{j=1}^{k}\sum_{i\in U'_j} w_i \geq \epsilon\sum_{j=1}^{k}\sum_{i\in V_j} w_i = \epsilon Wg(V)$$

Proof. (Lemma 4.1) Let $U_B(t)$ denote the set of jobs that WFB has at time t and $U_A(t)$ denote the set of jobs with *Opt*. Then from the preceding discussion, $Wg(U_B - U_A) \leq 1/\epsilon Wg(U_A)$. In other words, we have proved $Wg_{WFB}(t, 1+\epsilon) \leq (1 + 1/\epsilon)Wg_{Opt}(t, 1)$.

Thus we have proved the following theorem.

THEOREM 4.1. *WFB is a $(1+\epsilon)$-speed $(1 + 1/\epsilon)$-competitive non-clairvoyant algorithm for minimizing weighted flow time.*

5 Open problems

It will be extremely interesting to obtain a constant competitive online algorithm or to provide a better lower bound for the weighted flow time problem. The existence of a constant factor approximation algorithm is also open for the offline case.

A simpler proof of our result would be very interesting. More generally, can the proof technique be simplified and applied to other online algorithms.

Finally, in the non-clairvoyant weighted flow time case, it might be possible to prove a better competitive ratio than $(1 + 1/\epsilon)$, as done by Berman and Coulston [3] for the unweighted case.

Acknowledgements The authors would like to thank Avrim Blum, Chandra Chekuri and R. Ravi for very helpful discussions and comments.

References

[1] F. Afrati, E. Bampis, C. Chekuri, D. Karger, C. Kenyon, S. Khanna, I. Millis, M. Queyranne, M. Skutella, C. Stein, and M. Sviridenko. Approximation schemes for minimizing average weighted completion time with release dates. In *IEEE Symposium on Foundations of Computer Science*, pages 32–43, 1999.

[2] L. Becchetti, S. Leonardi, A. M. Spaccamela, and K. Pruhs. Online weighted flow time and deadline scheduling. In *RANDOM-APPROX*, pages 36–47, 2001.

[3] P. Berman and C. Coulston. Speed is more powerful than clairvoyance. *Nordic Journal of Computing*, 6(2):181–193, 1999.

[4] C. Chekuri and S. Khanna. Approximation schemes for preemptive weighted flow time. In *ACM Symposium on Theory of Computing (STOC)*, 2002.

[5] C. Chekuri, S. Khanna, and A. Zhu. Algorithms for weighted flow time. In *ACM Symposium on Theory of Computing (STOC)*, 2001.

[6] C. Chekuri, R. Motwani, B. Natarajan, and C. Stein. Approximation techniques for average completion time scheduling. In *In ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 609– 618, 1997.

[7] L. A. Hall, A. S. Schulz, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time: Offline and online algorithms. *Math. of Operations Research*, 22:513–544, 1997.

[8] S. Irani. Randomized weighted caching with two page weights. *Algorithmica*, 33:384–409, 2002.

[9] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM*, 47(4):617–643, 2000.

[10] J. Lenstra, A. Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[11] R. Motwani, S. Phillips, and E. Torng. Nonclairvoyant scheduling. *Theoretical Computer Science*, 130(1):17–47, 1994.

[12] L.E. Schrage. A proof of the optimality of the shortest processing remaining time discipline. *Operations Research*, 16:678–690, 1968.

[13] W. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3:59–66, 1956.

Appendix A

The $O(\log^2 P)$ algorithm of Chekuri *et al.* [5]: Let the set of jobs have weights drawn from the set $\{w_1, w_2, \ldots, w_{max}\}$. Assume $w_1 = 1$ and $w_i = (4(\log P + 1))^{i-1}$. For a job x, let $p_t(x)$ denote the remaining time of x at time t, and let $w(x)$ denote the weight of x. If $w(x) = w_i$, we say that the *weight index* of x is i. The *class* of a job is $\lfloor \log(p_t(x)/w(x)) \rfloor$. A job x belongs to the set $Q_{p,q}(t)$ if the weight index of x satisfies p and the class of x at time t satisfies q. $W_{p,q}(t)$ denotes the total weight of jobs in $Q_{p,q}(t)$. Their algorithm works as follows.

At each discrete time step t, choose a job in $Q(t)$ to execute by the following rules:

1. $i \leftarrow$ largest weight index in $Q(t)$.

2. $k \leftarrow$ smallest class in job of weight w_i in $Q(t)$.

3. If $W_{<i,<k(t)} \leq w_i$, schedule a job of weight w_i and class k (a job from $Q_{i,k}(t)$).

4. Else $i \leftarrow$ largest weight index in $Q(t)$ strictly less than i. Go to step 2.