

Non-clairvoyant Scheduling for Minimizing Mean Slowdown

N. Bansal ^{*} K. Dhamdhere [†] J. Könemann [‡] A. Sinha [§]

April 21, 2003

Abstract

We consider the problem of scheduling dynamically arriving jobs in a non-clairvoyant setting, that is, when the size of a job remains unknown until the job finishes execution. Our focus is on minimizing the mean *slowdown*, where the slowdown of a job (also known as stretch) is defined as the ratio of flow time to the size of the job.

We use *resource augmentation* in terms of allowing a faster processor to the online algorithm to make up for its lack of knowledge of job sizes. Our main result is that the Multi-level Feedback (MLF) algorithm [14, 16], used in the Windows NT and Unix operating system scheduling policies is an $(1+\epsilon)$ -speed $O((1/\epsilon)^5 \log^2 B)$ -competitive algorithm for minimizing mean slowdown non-clairvoyantly, when B is the ratio between the largest and smallest job sizes. In a sense, this provides a theoretical justification of the effectiveness of an algorithm widely used in practice. On the other hand, we also show that any $O(1)$ -speed algorithm, deterministic or randomized, is at least $\Omega(\min(n, \log B))$ competitive.

The motivation for resource augmentation is supported by an $\Omega(\min(n, B))$ lower bound on the competitive ratio without any speedup. For the static case, i.e. when all jobs arrive at time 0, we show that MLF is $O(\log B)$ competitive without any resource augmentation and also give a matching $\Omega(\log B)$ lower bound on the competitiveness.

Keywords: Scheduling, slowdown, online algorithms, non-clairvoyant algorithms, resource augmentation.

^{*}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.

[‡]Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA 15213. jochen@cmu.edu. This material is based upon work supported by the National Science Foundation under Grant No. 0105548.

[§]Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA 15213. asinha@andrew.cmu.edu

1 Introduction

1.1 Motivation

While scheduling algorithms in general have received a lot of interest in the past, most algorithms assume that all the information about a job is completely known when the job arrives. However, there are several situations where the scheduler has to schedule jobs without knowing the sizes of the jobs. This lack of knowledge, known as *non-clairvoyance*, is a significant impediment in the scheduler’s task, as one might expect. The study of non-clairvoyant scheduling algorithms was initiated by Motwani *et al.* [12].

We consider the problem of minimizing the total slowdown (also called stretch) non-clairvoyantly on a single processor with preemptions. This was posed as an important open problem by Becchetti *et al.* [3]. Slowdown or stretch of a job was first considered by Bender *et al.* [6]. It is the ratio of the flow time of a job to its size. Slowdown as a metric has received much attention lately [9, 17, 13, 5, 4, 1, 7], since it captures the notion of “fairness”. Note that a low slowdown implies that jobs are delayed in proportion to their size, hence smaller jobs are delayed less and large jobs are delayed proportionately more. Muthukrishnan *et al.* [13] first studied mean slowdown, and showed that the shortest remaining processing time (SRPT) algorithm achieves a competitive ratio of 2 for the single machine and 14 for the multiple machine case. Note that SRPT requires the knowledge of job sizes and hence cannot be used in the non-clairvoyant setting. Similarly, the various extensions and improvements [8, 1, 4, 7] to the problem have all been in the clairvoyant setting. Since clairvoyant scheduling does not accurately model many systems, there is significant interest in the non-clairvoyant version of slowdown.

As expected, non-clairvoyant algorithms usually have very pessimistic bounds (See for example [12]). A major advance in the study of non-clairvoyant algorithms was made by Kalyanasundaram and Pruhs [10], who proposed the model of *resource augmentation* where the online algorithm is compared against an adversary that has a slower processor. This provides a nice framework for comparing the performance of algorithms where the traditional competitive analysis gives a very pessimistic guarantee for all algorithms. Our analysis makes use of resource augmentation model.

1.2 Model

We consider a single machine scenario, where jobs arrive dynamically over time. Let \mathcal{J} denote the set of jobs. The size of a job $J_i \in \mathcal{J}$, denoted by p_i , is its total service requirement. However, p_i is not known by the scheduler at any time before the job completes, in particular p_i is not known when job i arrives. Obviously, p_i becomes known when the job completes. The time when a job arrives is known as its release date and is denoted by r_i . The completion time of

job J_i is denoted by c_i . The flow time of job J_i is defined as $f_i = c_i - r_i$ and its slowdown is $s_i = f_i/p_i$. Another way of writing slowdown of J_i is $\int_{r_j}^{c_j} \frac{1}{p_j} dt$. We are interested in minimizing the mean (or equivalently total) slowdown.

Traditionally, an online algorithm is said to be c -competitive if the worst case ratio (over all possible inputs) of the performance of the online algorithm is no more than c times the performance of the optimum offline adversary. In the resource augmentation model [10], we say that an algorithm is s -speed c -competitive if it uses an s times faster processor than the optimum algorithm and produces a solution that has competitive ratio no more than c against the optimal algorithm with no speedup.

1.3 Results

1. *General Case:* The main contribution of this paper is to show that MLF is $O((1/\epsilon)^5 \log^2 B)$ -competitive with a $(1+\epsilon)$ -speedup, for any fixed $\epsilon > 0$. Here B is the ratio of the maximum to the minimum job size. We note that MLF does not need the knowledge of B and is fully online.
2. *Need for Resource Augmentation:* It is not hard to see that in the absence of an upper bound on job sizes, no algorithm for minimizing mean slowdown can be $\Omega(n)$ competitive. Surprisingly, it turns out that no algorithm (deterministic or randomized) can be $o(n/k)$ -competitive even with a k -speedup. In particular, we need at least $\Omega(n)$ speedup to be competitive.
3. The above lower bounds require instances where the range of job sizes varies exponentially in n . In a more realistic scenario, when job size ratios are bounded by B , we show that in the absence of speedup, any algorithm (deterministic or randomized) is $\Omega(B)$ competitive. Moreover, we show that even with a factor k speedup any algorithm (deterministic or randomized) is at least $\Omega(\log B/k)$ competitive. Note that the performance our algorithm matches the lower bound upto a $\log B$ factor in the competitive ratio.
4. *Static Case:* When all the requests arrive at time 0, we settle the question exactly. We show that in this case, MLF is $O(\log B)$ competitive without any speed up. We also show matching $\Omega(\log B)$ lower bound on the competitive ratio of any deterministic or randomized algorithm, hence implying that the performance guarantee provided by MLF is tight.

We also stress that our algorithm for the general case is fast and easy to implement, which is a useful advantage when it comes to putting it to work in real systems, say a web server. In

fact, it can be implemented in $O(n)$ space, with only $O(\log B)$ preemptions per job. It needs only $O(1)$ work per preemption.

1.4 Relations to Other Problems

Minimizing total slowdown can be thought of as a special case of minimizing total weighted flow time, where the weight of a job is inversely proportional to its size. However, what makes the problem interesting and considerably harder in a non-clairvoyant setting is that the sizes (hence weights) are not known. Hence not only does the online scheduler have no idea of job sizes, it also has no idea as to which job is more important (has a higher weight).

An interesting aspect of slowdown is that neither resource augmentation nor randomization seems to help in the general case. This is in sharp contrast to weighted flow time where a $(1 + \epsilon)$ speed gives a $(1 + \frac{1}{\epsilon})$ competitive algorithm [2]. Similarly, for flow time there is a lower bound of $n^{1/3}$ for any deterministic algorithm, whereas using randomization, algorithms which are $O(\log n)$ competitive can be obtained [11, 3]. In contrast, even with randomization and a k speed-up the slowdown problem is at least $\Omega(n/k)$ competitive.

In terms of proof techniques, all previous techniques using resource augmentation [10, 5, 2] relied on *local competitiveness*¹. For non-clairvoyant slowdown, proving local competitiveness is unlikely to work: it could easily be the case that the optimal clairvoyant algorithm has a few jobs of size B , while the non-clairvoyant online algorithm is left with a few jobs of size 1, thus being $\Omega(B)$ uncompetitive locally.

Our main idea is to define a new metric, which we call the *inverse work* metric. We use this metric to connect the performance of the online algorithm to the offline algorithm. We also use resource augmentation in a novel way which can potentially find uses elsewhere.

2 The MLF algorithm

We first describe a slight variation of MLF algorithm. For any fixed $\epsilon > 0$, we show that it is $(1 + \epsilon)$ -speed $O((1/\epsilon)^5 \log^2 B)$ -competitive. Here B is the ratio of maximum to minimum job size. Our algorithm does not need to know B .

¹Intuitively, an algorithm is locally competitive if the cost incurred in objective in each unit step is within a constant factor of that incurred by optimal algorithm. Local competitiveness is both necessary and sufficient for global competitiveness [5] when there is no speedup.

2.1 Algorithm Description

A job is said to be *active* at time t , if it is released by that time but not completed. Denote by $x_j(t)$ the amount of work that has been done on job J_j by time t . Let $l_i = \epsilon((1 + \epsilon)^i - 1)$ for $i = 0, 1, \dots$

Partition the active jobs into a set of queues Q_0, Q_1, \dots . Here, Q_0 is the lowest queue, Q_1 is the next lowest and so on. A job j is in the queue Q_k at time t , if the work done $x_j(t) \in [l_k, l_{k+1})$. MLF always works on the earliest arriving job in the lowest non-empty queue. Thus, a job is placed in Q_0 upon its arrivals and moves up the queues during its execution until it finishes.

Thus amount of work done on a job in queue k of MLF is exactly $l_{k+1} - l_k$. We call this q_k . And it is given by $q_k = \epsilon^2(1 + \epsilon)^k$. Observe that $l_k = q_0 + q_1 + \dots + q_{k-1}$.

For a scheduling algorithm A , we use $A(s, \mathcal{J})$ to denote the slowdown of the algorithm A on instance \mathcal{J} when provided with an s speed processor.

We now state our main result of this paper. The proof will be given in the next subsection.

Theorem 1 *For a fixed $\epsilon > 0$, MLF algorithm as defined above is $(1 + \epsilon)$ -speed $O((\frac{1}{\epsilon})^5 \log^2(B))$ -competitive algorithm.*

2.2 Analysis

Given the original scheduling instance \mathcal{J} , we want to modify it so that sizes of all the jobs are rounded up to the queue thresholds in MLF. We construct a modified instance \mathcal{K} as follows. Consider a job $J_j \in \mathcal{J}$ and let i be the smallest integer such that $p_j \leq \epsilon((1 + \epsilon)^i - 1)$. Corresponding to the job J_j , we add a job (also denoted by J_j) in \mathcal{K} with the same release time. We set its size to be $\epsilon((1 + \epsilon)^i - 1)$. Observe that, size of each job in \mathcal{K} is l_k for some k , identical to the threshold for one of MLF queues.

Observation 1 *In the transformation from \mathcal{J} to \mathcal{K} , size of a job J_j doesn't decrease. Moreover, it increases by at most a factor of $(1 + \epsilon)^2$.*

Note that, algorithm MLF has the property that, if we increase size of a job, it will not decrease the completion time of any job. The following lemma shows us that the transformation from instance \mathcal{J} to \mathcal{K} did not increase the slowdown of MLF by a lot.

Lemma 1 $MLF(s, \mathcal{J}) \leq (1 + \epsilon)^2 MLF(s, \mathcal{K})$

Proof: For a job J_j , let f_j be it's flowtime in instance \mathcal{J} and f'_j be the flowtime in instance \mathcal{K} . Also let p'_j denote it's size in instance \mathcal{K} .

Then we know from Observation 1, that $f_j \leq f'_j$. Moreover, $p'_j \leq (1 + \epsilon)^2 p_j$. Combining these,

$$f_j/p_j \leq (1 + \epsilon)^2 f'_j/p'_j$$

Adding slowdown contributions for all jobs in the instance \mathcal{J} we get the result. \blacksquare

Thus, instead of working on the original input instance \mathcal{J} , we will work with the modified instance \mathcal{K} , in which all job sizes are rounded up to the queue thresholds of MLF.

We now define a new metric called the “inverse work” metric as an upper bound for slowdown.

Definition 1 Consider any scheduling instance \mathcal{I} , and an algorithm A operating on it. Let J_j be a job and if at time t , the amount of work done on J_j satisfies the relation $\epsilon((1 + \epsilon)^i - 1) \leq x_j(t) < \epsilon((1 + \epsilon)^{i+1} - 1)$, then define $w_j(t) = \epsilon((1 + \epsilon)^{i+1} - 1)$. The inverse work of job J_j is defined as: $\int_{r_j}^{c_j} \frac{1}{w_j(t)} dt$. And the total inverse work of algorithm A is $\sum_{j \in \mathcal{I}} \int_{r_j}^{c_j} \frac{1}{w_j(t)} dt$.

We will denote the inverse work of algorithm A on input instance \mathcal{I} , using an s speed processor by $A'(s, \mathcal{I})$.

The main intuition of analysis is the following. A non-clairvoyant algorithm must spend some processing time probing for jobs of small sizes. However, this can cause it to postpone a small job while working on a large job. A clairvoyant algorithm on the other hand, has the advantage that it can postpone working on a large job for a long time. The inverse work metric takes care of essentially that, it captures the advantage of “probing”. Another property of this metric is that a good schedule for slowdown can be converted into a good schedule for inverse work, with some speed up. These two ideas form the crux of our analysis. The details of the argument are given below.

First we will show that on the input instance \mathcal{K} that we constructed above, we can bound the slowdown in terms of inverse work.

Lemma 2 For any algorithm A on the instance \mathcal{K} , the slowdown is upper bounded by the inverse work.

$$A(s, \mathcal{K}) \leq A'(s, \mathcal{K})$$

Proof: Consider a job J_j . At a time instance t , such that $r_j \leq t \leq c_j$, the work done on the job is less than its size. Also, in the instance \mathcal{K} , we have $p_j = \epsilon((1 + \epsilon)^i - 1)$ for some i . Hence $w_j(t) \leq p_j$ at all times $t \leq c_j$. Hence, contribution of job J_j in slowdown can be bounded as

$$\int_{r_j}^{c_j} \frac{1}{p_j} dt \leq \int_{r_j}^{c_j} \frac{1}{w_j(t)} dt$$

Summing up over all jobs in the instance \mathcal{K} gives the inequality. ■

From the input instance \mathcal{K} we create a new instance \mathcal{L} by replacing each job J_j of size $\epsilon((1 + \epsilon)^k - 1)$ by jobs $J_{j_0}, J_{j_1}, \dots, J_{j_{k-1}}$, with the sizes q_0, q_1, \dots, q_{k-1} respectively. The release dates remain same. Note that the sizes of the jobs $J_{j_0}, J_{j_1}, \dots, J_{j_{k-1}}$ add up to the size of job J_j , since $size(J_j) = l_k = q_0 + q_1 + \dots + q_{k-1}$.

We view MLF algorithm working on instance \mathcal{K} as the algorithm SJF (Shortest Job First) running on instance \mathcal{L} . The following observation makes the connection clear.

Observation 2 *At any time t , the algorithm SJF with input instance \mathcal{L} is working on a job $J_{j_b} \in \mathcal{L}$ iff MLF with input \mathcal{K} is working on the job $J_j \in \mathcal{K}$ in queue b at the same time. In particular, SJF has completed the jobs $J_{j_0}, J_{j_1}, \dots, J_{j_{b-1}}$.*

This gives us a way of relating inverse work of MLF on instance \mathcal{K} to the slowdown of SJF on instance \mathcal{L} .

Lemma 3 *The slowdown of SJF on instance \mathcal{L} dominates the inverse work of MLF on instance \mathcal{K}*

$$MLF'(s, \mathcal{K}) \leq SJF(s, \mathcal{L})$$

Proof: Consider a job $J_j \in \mathcal{K}$. Suppose, at the time t , we have $w_j(t) = \epsilon((1 + \epsilon)^{i+1} - 1)$ for some $i \geq 0$. Then, the contribution to the inverse work of MLF by the job J_j at time t is $\frac{1}{w_j(t)}$. i

However, work done by MLF on job J_j is at least $\epsilon((1 + \epsilon)^i - 1)$. Thus J_j is in the queue Q_i of MLF. From observation 2, we know that SJF has finished the jobs $J_{j_0}, J_{j_1}, \dots, J_{j_{i-1}}$. Thus slowdown contribution by jobs $J_{j_i}, J_{j_{i+1}}, \dots$ in SJF is at least the contribution by the job J_{j_i} . Size of J_{j_i} is $\epsilon^2(1 + \epsilon)^i$. Thus its contribution is $\frac{1}{\epsilon^2(1 + \epsilon)^i}$.

Now it is easy to observe that $\frac{1}{w_j(t)} = \frac{1}{\epsilon((1 + \epsilon)^{i+1} - 1)} \leq \frac{1}{\epsilon^2(1 + \epsilon)^i}$. Thus for the job J_j , it's inverse work contribution to MLF is at most the slowdown contribution by corresponding jobs J_{j_0}, J_{j_1}, \dots to SJF. Hence aggregating over all jobs in \mathcal{K} we get the result. ■

Thus by lemma 1 & 2

$$MLF(s, \mathcal{J}) \leq (1 + \epsilon)^2 SJF(s, \mathcal{L}) \tag{1}$$

We now relate the slowdown on SJF algorithm on instance \mathcal{L} back to the slowdown for instance \mathcal{K} . For this purpose we modify \mathcal{J} further. Let \mathcal{K}' denote the following modification on instance \mathcal{K} . Add ϵ to the size of each job. The size of each job in \mathcal{K} is $\epsilon((1 + \epsilon)^i - 1)$ for some

$i \geq 1$. Now size of each job in \mathcal{K}' is $\epsilon(1 + \epsilon)^i$. Once again, we note here that size of each job $J_j \in \mathcal{J}$ does not decrease in \mathcal{K}' and does not increase by more than a factor of $(1 + \epsilon)^2$.

Let $\mathcal{L}(k)$ denote the instance obtained by multiplying each job size in \mathcal{K}' by $\epsilon/(1 + \epsilon)^k$. Next, we remove from $\mathcal{L}(k)$ any job whose size is smaller than ϵ^2 .

We claim that, $\mathcal{L} = \mathcal{L}(1) \cup \mathcal{L}(2) \cup \dots$. To see this, consider a job $J_j \in \mathcal{K}'$ of size $\epsilon(1 + \epsilon)^i$. Then $\mathcal{L}(1)$ contains the corresponding job $J_{j_{i-1}}$ of size $\epsilon/(1 + \epsilon) \cdot \epsilon(1 + \epsilon)^i = \epsilon^2(1 + \epsilon)^{i-1} = q_{i-1}$. Similarly, $\mathcal{L}(2)$ contains the job $J_{j_{i-2}}$ of size q_{i-2} and so on. Thus \mathcal{L} is exactly $\mathcal{L}(1) \cup \mathcal{L}(2) \cup \dots$. In short, \mathcal{L} is the union of scaled down copies of \mathcal{K}' .

Now the idea is to construct a schedule for $\mathcal{L}(k)$ using SJF schedule for \mathcal{K}' . This will be achieved in the next couple of lemmas.

Lemma 4 *Let $f_{SJF}(s, j)$ denote the flow time of job j under SJF with a s -speed processor. Then under SJF with an $s \geq 1$ speed processor, we have $f_{SJF}(s, j) \leq (1/s)f_{SJF}(1, j)$.*

Proof: Let $x_j(u, 1)$ denote the work done on job j , after u units of time since it arrived, under SJF using a 1 speed processor. Similarly, let $x_j(u, s)$ denote the work done on job j , after u units of time since it arrived, under SJF using an s speed processor. We will show a stronger invariant that for all jobs j and all times t , $x_j((t - r_j)/s, s) \geq x_j(t - r_j, 1)$. Notice that this stronger invariant trivially implies the result of the lemma.

Consider some instance where this condition is violated. Let j be the job and t be the earliest time for which $x_j((t - r_j)/s, s) < x_j(t - r_j, 1)$. Clearly, the speed s processor (SJF(s)) is not working on j at time t , due to minimality of t . Thus, SJF(s) is working on some other smaller job j' . Since SJF(1) is not working on j' , it has already finished j' by some time $t' < t$. However, this means that $x_{j'}((t' - r_{j'})/s, s) < x_{j'}(t' - r_{j'}, 1)$, which contradicts the minimality of t . ■

Lemma 5 *Let $\mathcal{L}(k)$ be the instance defined as above. Then if $x \geq \epsilon(1 + \epsilon)^{-k}$, we have*

$$SJF(x \cdot s, \mathcal{L}(k)) \leq \frac{1}{x} SJF(s, \mathcal{K}')$$

Proof: A job $J_{j_k} \in \mathcal{L}(k)$ is a copy of the job $J_j \in \mathcal{K}'$ with the size scaled by $\epsilon(1 + \epsilon)^{-k}$. Thus, if we run SJF on $\mathcal{L}(k)$ with an $\epsilon(1 + \epsilon)^{-k} \cdot s$ speed processor, it will correspond to running SJF on \mathcal{K}' with an s speed processor. Hence the flow time of $J_{j_k} \in \mathcal{L}(k)$ will be same as that of $J_j \in \mathcal{K}'$.

From Lemma 4, if we use $x \cdot s$ speed instead of $\epsilon(1 + \epsilon)^{-k} \cdot s$, for SJF on $\mathcal{L}(k)$, then the flow time of J_{j_k} will be at most $\epsilon(1 + \epsilon)^{-k}/x$ times smaller. However, size of J_{j_k} is $\epsilon(1 + \epsilon)^{-k}$ times

smaller than $J_j \in \mathcal{K}'$. Thus the slowdown of J_{j_k} is at most $\epsilon(1 + \epsilon)^{-k}/x \cdot 1/(\epsilon(1 + \epsilon)^{-k}) = 1/x$ times that of J_j . This proves the result. ■

Now we will relate the optimal offline schedule with the schedule produced by SJF. For an input instance \mathcal{I} , we will use $OPT(s, \mathcal{I})$ to denote the slowdown incurred by optimal schedule for instance \mathcal{I} using an s speed processor.

Lemma 6 *For the input instances \mathcal{L} and \mathcal{K}' as defined above, we have*

$$OPT((1 + \epsilon) \cdot s, \mathcal{L}) \leq O\left(\frac{1}{\epsilon}(\log_{1+\epsilon}^2 B)\right) SJF(s, \mathcal{K}')$$

Proof: Now we construct a schedule A for \mathcal{L} by superimposing the SJF schedules for the instances $\mathcal{L}(k)$'s. The jobs in $\mathcal{L}(k)$ are run with a speed x_k using the schedule given by SJF. Note that the maximum job size is B , so we need to look at instances $\mathcal{L}(k)$ up to $k = \log_{1+\epsilon}(B/\epsilon)$.

We set $x_i = \epsilon(1 + \epsilon)^{-i}$ for $i = 1, 2, \dots, \log_{1+\epsilon} \log_{1+\epsilon}(B/\epsilon)$ and $x_i = \epsilon/\log_{1+\epsilon}(B/\epsilon)$ for $i > \log_{1+\epsilon} \log_{1+\epsilon}(B/\epsilon)$. Notice that the total speed required by A is $\sum_{i=1}^{\log_{1+\epsilon}(B/\epsilon)} x_i \leq 1 + \epsilon$.

By Lemma 5, the slowdown of schedule A on $\mathcal{L}(k)$ will be at most $(\frac{1}{x_i})SJF(1, \mathcal{K}')$. Thus we can bound the total slowdown of A on the instance \mathcal{L} as follows

$$A((1 + 2\epsilon) \cdot s, \mathcal{L}) \leq \sum_{i=1}^{\log_{1+\epsilon}(B/\epsilon)} \left(\frac{1}{x_i}\right) SJF(s, \mathcal{K}') \leq \left(\frac{1}{\epsilon}\right) \log_{1+\epsilon}^2(B/\epsilon) SJF(s, \mathcal{K}')$$

since we can bound the sum $\sum_i \frac{1}{x_i}$ by $(\frac{1}{\epsilon}) \log_{1+\epsilon}^2(B/\epsilon)$. Lastly, since OPT is at least as good as A, we get the result. ■

We need to bound the slowdown of SJF on an instance by the slowdown of OPT on the same instance. We use a more general result due to Becchetti *et al.* [5] for this purpose. They show that for the problem of minimizing weighted flow time, the greedy algorithm *Highest Density First* (HDF) is $(1 + \epsilon)$ -speed $(1 + 1/\epsilon)$ -competitive algorithm. The HDF algorithm works on the job with highest weight to size ratio at any time. Since slowdown is a special case of weighted flow time with weights equal to the reciprocal of size, we get that for any input instance \mathcal{I} ,

$$SJF((1 + \epsilon) \cdot s, \mathcal{I}) \leq \left(1 + \frac{1}{\epsilon}\right) OPT(s, \mathcal{I}) \tag{2}$$

We use this fact for the quantities in Lemma 6 to get the following:

$$SJF((1 + \epsilon)^3 \cdot s, \mathcal{L}) \leq \left(1 + \frac{1}{\epsilon}\right) OPT((1 + \epsilon)^2 \cdot s, \mathcal{L})$$

and

$$O\left(\frac{1}{\epsilon}\right)^2(\log_{1+\epsilon}^2 B) S J F((1 + \epsilon) \cdot s, \mathcal{K}') \leq O\left(\frac{1}{\epsilon}\right)^3(\log_{1+\epsilon}^2 B) O P T(s, \mathcal{K}')$$

These two together give us:

$$S J F((1 + \epsilon)^3 \cdot s, \mathcal{L}) \leq O\left(\frac{1}{\epsilon}\right)^3(\log_{1+\epsilon}^2 B) O P T(s, \mathcal{K}') \quad (3)$$

Proof:(of Theorem 1) Given an input instance \mathcal{J} , we constructed the input instance \mathcal{L} and showed in Equation 1 that

$$M L F((1 + \epsilon)^5, \mathcal{J}) \leq (1 + \epsilon)^2 S J F((1 + \epsilon)^5, \mathcal{L}) \quad (4)$$

Combining this with Equation 3, we get

$$M L F((1 + \epsilon)^5, \mathcal{J}) \leq O\left(\frac{1}{\epsilon}\right)^3(\log_{1+\epsilon}^2 B) O P T((1 + \epsilon)^2, \mathcal{K}') \quad (5)$$

Notice that each job $J_j \in \mathcal{K}'$ has size at most $(1 + \epsilon)^2$ times its size in the original instance \mathcal{J} . Thus we trivially have that,

$$O P T((1 + \epsilon)^2, \mathcal{K}') \leq O P T(1, \mathcal{J}) \quad (6)$$

Equations 5 and 6 together give us that

$$M L F((1 + \epsilon)^5, \mathcal{J}) \leq O\left(\frac{1}{\epsilon}\right)^3(\log_{1+\epsilon}^2 B) O P T(1, \mathcal{J})$$

or equivalently that,

$$M L F((1 + \epsilon)^5, \mathcal{J}) \leq O\left(\frac{1}{\epsilon}\right)^5(\log^2 B) O P T(1, \mathcal{J})$$

■

3 Lower bounds

We now give lower bounds which motivate our algorithm in the previous section, in particular the need for bounded job sizes and the need for resource augmentation. Moreover, all of the lower bounds also hold if we allow the algorithm to be randomized.

3.1 Bounded job sizes without resource augmentation

Without resource augmentation the performance of any non-clairvoyant algorithm is really bad, even when job sizes are bounded. We show that any non-clairvoyant algorithm *without speedup*, deterministic or randomized is at least $\Omega(B)$ competitive.

Theorem 2 *Any non-clairvoyant algorithm, deterministic or randomized is at least $\Omega(B)$ competitive for mean slowdown, where B is the ratio of job sizes.*

Proof: Consider an instance where nB jobs of size 1, and n jobs of size B , arrive at time $t = 0$. At time $t = nB$, the adversary gives a stream of m jobs of size 1 every unit of time.

The optimum algorithm finishes all jobs of size 1 by nB and continues to work on the stream of size 1 jobs. The slowdown incurred due to jobs of size B is at most $n(2nB + m)/B$ and due to the jobs of size 1 is $nB + m$. The deterministic non-clairvoyant algorithm, on the other hand, has to spend at least one unit of processing on each job to determine if it has size 1 or B . Thus, by choosing the first n jobs on which the non-clairvoyant algorithm works at least 1 unit to be of size B , it can be made to have at least n jobs of size 1 remaining at time $t = nB$. For the next m units after time $t = nB$, there will be at least n jobs of size 1. Thus, total slowdown incurred is at least nm . Choosing $n > B$ and $m > nB$, it is easy to see that the competitive ratio is at least $\Omega(B)$.

For the randomized case, we use Yao's Lemma, and the input instance consists of $nB + n$ jobs, n of which are chosen randomly and have size B while the rest have size 1. Again, since any non-clairvoyant algorithm has to spend at least 1 unit of processing to distinguish between at job of size 1 and B , it follows that by time $t = nB$, the algorithm will have at least $\frac{B}{B+1}n \approx n$ jobs of size 1 remaining in expectation. Thus the result follows. ■

3.2 Lower bound with bounded size and resource augmentation

We now consider lower bounds when resource augmentation is allowed. We first consider a static (all jobs arrive at time $t = 0$) scheduling instance, and give an $\Omega(\log B)$ lower bound without resource augmentation. While this in itself is weaker than Theorem 2, the scheduling instance being static implies that even resource augmentation by k times can only help by a factor of k .

Lemma 7 *No deterministic or randomized algorithm can have performance ratio better than $\Omega(\log B)$ for a static scheduling instance, where B is the ratio of the largest to the smallest job size.*

Proof: We consider an instance with $\log B$ jobs $j_0, \dots, j_{\log B}$ such that job j_i has size 2^i .

We first look at how SJF behaves on this problem instance. The total slowdown for SJF is $\sum_{i=0}^{\log B} \frac{1}{2^i} \left(\sum_{j=0}^i 2^j \right) = O(\log B)$. This basically follows from the fact that SRPT has to finish jobs j_1, \dots, j_{i-1} before it finishes j_i by the definition of our instance.

Now we show that for any non-clairvoyant deterministic algorithm A , the adversary can force the total slowdown to be $\Omega(\log^2 B)$. The rough idea is as follows: We order the jobs of our instance such that for all $0 \leq i \leq \log B$, A spends at least 2^i work on jobs $j_{i+1}, \dots, j_{\log B}$ before it finishes job j_i . In this case, the theorem follows because the total slowdown of A on the given instance is

$$\sum_{i=1}^{\log B} \frac{1}{2^i} (\log(B) - i + 1) 2^i = \Omega(\log^2(B)).$$

It remains to show that we can order the jobs in such a way. Since A is a deterministic algorithm that does not use the size of incoming jobs, we can determine the order in which jobs receive a total of at least 2^i work by A . We let j_i be the $(\log(B) - i + 1)^{th}$ job that receives 2^i work for all $0 \leq i \leq \log B$. It is clear that this yields the claimed ordering.

The example can be randomized to prove that even a randomized algorithm has mean slowdown no better than $\Omega(\log B)$. The idea is to assume that the instance is a random permutation of the jobs $j_0, j_1, \dots, j_{\log B}$. Then to finish job j_i , the scheduler has to spend at least 2^i work on at least half of $j_{i+1}, \dots, j_{\log B}$ (in expectation). Thus its expected slowdown is $\frac{1}{2}(\log B - i + 1)$ and the total slowdown is $\Omega(\log^2 B)$. We now use Yao's Minimax Lemma to obtain the result. ■

As the input instance in Lemma 7 is static, a k -speed processor can at most improve all the flow times by a factor of k . Hence the mean slowdown can go down by the same factor. This gives us the following theorem.

Theorem 3 *Any k -speed deterministic or randomized, non-clairvoyant algorithm has an $\Omega(\log B/k)$ competitive ratio for minimizing mean slowdown, in the static case (and hence in the online case).*

3.3 Scheduling with general job sizes

The previous result also implies the following lower bound when job sizes could be arbitrarily large. In particular, we can choose the job sizes to be $1, 2, \dots, 2^n$ which gives us the following theorem.

Theorem 4 *Any k -speed non-clairvoyant algorithm, either deterministic or randomized, has $\Omega(n/k)$ performance ratio for minimizing mean slowdown.*

Theorems 2 and 4 show that in the absence of resource augmentation and bounded job sizes achieving any reasonable guarantee on the competitive ratio is impossible. This motivates our model in Section 2, where we assume bounded job sizes and use resource augmentation.

4 Static scheduling

Static scheduling is usually substantially easier than the usual dynamic scheduling (where jobs have arbitrary release times), and the same is the case here. We do not need resource augmentation here, and we show that the *MLF* is $O(\log B)$ competitive, hence matching the lower bound shown in the previous section.

4.1 Optimal Clairvoyant Algorithm

In that static case, it follows easy from Smith's rule [15] that SJF is the optimal algorithm for minimizing total slowdown. We now show a rather general lower bound on the mean slowdown under SJF on any input instance.

Lemma 8 *For any scheduling instance with n jobs all arriving at time 0, SJF has $\Omega(n/\log B)$ mean slowdown.*

Proof: We first show that we can assume without loss of generality that we can consider an input instance where the job sizes are powers of 2. To see this, given instance I , We lower bound the total slowdown of SJF as follows: for a job of size x , we require SJF to work only $2^{\lceil \lg x \rceil}$ amount in order to finish the job and we divide the flow time by $2^{\lceil \lg x \rceil}$ to get slowdown. Thus, we can round down all the job sizes to a power of 2 and have the new total slowdown within a factor of 2 of the total slowdown of original instance.

Now we have $x_1, x_2, \dots, x_{\log B}$ jobs of sizes $2, 4, \dots, B$ respectively. We also have $\sum x_i = n$. Now, as there are at least x_i jobs of size 2^i , the average flow time of a job of size 2^i under SJF is at least $x_i 2^i / 2$, and hence its slowdown is at least $x_i / 2$. Since there are x_i such jobs, it follows that the contribution to the total slowdown by jobs of size 2^i is at least $(1/2)x_i^2$. Thus the total slowdown is at least $\sum_i (1/2)x_i^2 \geq 1/(2 \log B)(\sum_i x_i)^2$ (by Cauchy-Schwarz) which is $\Omega(n^2/\log B)$. Thus the mean slowdown for SJF is $\Omega(n/\log B)$. ■

4.2 Competitiveness of MLF

Lemma 9 *For a scheduling instance with n jobs, MLF has a $O(n)$ mean slowdown.*

Proof: This follows trivially, since for any job of size 2^i , the job waits at most for n other jobs to receive at processing of at most 2^i . ■

Combining the results of Lemmas 8 and 9, we get the following result:

Theorem 5 *For static scheduling with bounded job sizes, the MLF algorithm is $O(\log B)$ -competitive for minimizing mean slowdown.*

5 Open questions

The only “gap” in our paper is the discrepancy between the upper and lower bounds for the main problem of non-clairvoyant scheduling to minimize mean slowdown. It would be interesting to close this.

Acknowledgments

The authors would like to thank Avrim Blum, Moses Charikar, Kirk Pruhs and R. Ravi for useful discussions.

References

- [1] B. Awerbuch, Y. Azar, S. Leonardi, and O. Regev. Minimizing the flow time without migration. In *ACM Symposium on Theory of Computing*, pages 198–205, 1999.
- [2] N. Bansal and K. Dhamdhere. Minimizing weighted flow time. In *14th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2003.
- [3] L. Becchetti and S. Leonardi. Non-clairvoyant scheduling to minimize the average flow time on single and parallel machines. In *ACM Symposium on Theory of Computing (STOC)*, pages 94–103, 2001.
- [4] L. Becchetti, S. Leonardi, and S. Muthukrishnan. Scheduling to minimize average stretch without migration. In *Symposium on Discrete Algorithms*, pages 548–557, 2000.
- [5] Luca Becchetti, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Kirk R. Pruhs. On-line weighted flow time and deadline scheduling. *Lecture Notes in Computer Science*, 2129:36–47, 2001.

- [6] M. Bender, S. Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 270–279, 1998.
- [7] M. Bender, S. Muthukrishnan, and R. Rajaraman. Improved algorithms for stretch scheduling. In *13th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2002.
- [8] C. Chekuri, S. Khanna, and A. Zhu. Algorithms for weighted flow time. In *ACM Symposium on Theory of Computing (STOC)*, 2001.
- [9] M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in web servers. In *USENIX Symposium on Internet Technologies and Systems*, 1999.
- [10] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM*, 47(4):617–643, 2000.
- [11] Bala Kalyanasundaram and Kirk Pruhs. Minimizing flow time nonclairvoyantly. In *IEEE Symposium on Foundations of Computer Science*, pages 345–352, 1997.
- [12] R. Motwani, S. Phillips, and E. Torng. Nonclairvoyant scheduling. *Theoretical Computer Science*, 130(1):17–47, 1994.
- [13] S. Muthukrishnan, R. Rajaraman, A. Shaheen, and J. Gehrke. Online scheduling to minimize average stretch. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 433–442, 1999.
- [14] G. Nutt. *Operating system projects using Windows NT*. Addison Wesley, 1999.
- [15] W. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3:59–66, 1956.
- [16] A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.
- [17] H. Zhu, B. Smith, and T. Yang. Scheduling optimization for resource-intensive web requests on server clusters. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 13–22, 1999.