Non-clairvoyant Scheduling for Minimizing Mean Slowdown

N. Bansal[1], K. Dhamdhere[2], J. Könemann[3], and A. Sinha[4]

[1] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA,
nikhil@cs.cmu.edu
[2] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA,
kedar@cs.cmu.edu
[3] Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA 15213, USA,
jochen@cmu.edu
[4] Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA 15213, USA,
asinha@andrew.cmu.edu

Abstract. We consider the problem of scheduling jobs online non-clairvoyantly, that is, when job sizes are not known. Our focus is on minimizing mean *slowdown*, 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 an $O(1)$-speed $O(\log^2 B)$-competitive algorithm for minimizing mean slowdown non-clairvoyantly, when B is the ratio between the largest and smallest job sizes. On the other hand, we show that any $O(1)$-speed algorithm, deterministic or randomized, is at least $\Omega(\log B)$ competitive.
The motivation for bounded job sizes is supported by an $\Omega(n)$ lower bound for arbitrary job sizes, where n is the number of jobs. Furthermore, a lower bound of $\Omega(B)$ justifies the need for resource augmentation even with bounded job sizes. For the static case, i.e. when all jobs arrive at time 0, we give an $O(\log B)$ competitive algorithm which does not use resource augmentation and a matching $\Omega(\log B)$ lower bound on the competitiveness.

1 Introduction

1.1 Motivation

While scheduling algorithms in general have received a lot of interest in the past, most algorithms assume that the input is completely known. 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, 15, 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. The size of a job i, 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 flow time of a job is the difference of the times when the job completes (c_i) and the time when it arrives, i.e. its release date (r_i). The slowdown of a job is the ratio of its flow time to its size. 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:* Our main contribution is an algorithm for minimizing mean slowdown non-clairvoyantly which is $O(\log^2 B)$-competitive with an $O(1)$-speedup, where B is the ratio of the maximum to the minimum job size. We

also note that our algorithm does not need to know B and is fully online. We will use the existence of B only in our analysis.
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 give an $O(\log B)$ competitive algorithm that does not use any speed up and a matching $\Omega(\log B)$ lower bound on the competitive ratio of any deterministic or randomized algorithm.

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 $O(n)$ time.

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*[5]. For non-clairvoyant slowdown, proving local competitiveness is unlikely to work: it could easily be the case

[5] Intuitively, 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.

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 extremely 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 Scheduling for mean slowdown

We first present an $O(1)$-speed, $O(\log^2 B)$-competitive algorithm for minimizing mean slowdown non-clairvoyantly. The comparison is made against the best possible clairvoyant scheduling algorithm. Here B is an upper bound on the ratio between the biggest and smallest job size. We will show later that we cannot design significantly better algorithms unless we have bounded job sizes, and resource augmentation.

Let J be a scheduling instance on n jobs with processing times $p_1, \ldots, p_n$ and release times $r_1, \ldots, r_n$. For a scheduling algorithm A, we let $S(A, s, J)$ denote the slowdown when A is provided with an s speed processor. Let $Opt_S(s, J)$ denote the slowdown of the algorithm which has optimal slowdown on the instance J using an s speed processor.

Let J' be a scheduling instance on the same jobs as J but with job sizes $p'_i = 2^{\lceil \log_2 p_i \rceil}$ for all $1 \leq i \leq n$. It is easy to see that

$$Opt_S(2s, J') \leq Opt_S(s, J) \tag{1}$$

Imagine an algorithm with a 2-speed processor that simply mimics the optimum algorithm for the speed 1 processor. It can possibly sit idle when the speed 1 processor is still working on a job which the speed two processor has already finished. And hence slowdown on rounded-up instance of optimal algorithm will be as good as the slowdown of optimal algorithm with a speed 1 processor. In fact, with a speed-up 2, we can assume that job sizes are of the form $2^i - 1$.

Secondly, using a speed up of 2 we also know that *Shortest Job First* (SJF) is a 2-competitive algorithm for slowdown. This follows from a more general result due to Becchetti *et al.* [5]. They show that for the problem of minimizing weighted flow time, the greedy algorithm *Highest Density First* (HDF) is a $(1+\epsilon)$-speed, $(1 + 1/\epsilon)$ competitive algorithm. The HDF algorithm works on the job with the 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,

$$S(SJF, 2s, J) \leq 2Opt_S(s, J) \tag{2}$$

2.1 Algorithm description

The input consists of jobs arriving over a period of time. We assume that the smallest job has size 1. More than one job may arrive at a given instant of time.

We invoke Equation 1 to assume that all jobs have sizes which are of the form $2^i - 1$.

We use the *Multi-Level Feedback* (MLF) algorithm [14] as our scheduling algorithm. MLF scheduling algorithm is used by Unix operating system. In MLF, jobs are queued in different queues according to the amount of work done on them. Queue i holds jobs which have between $2^i - 1$ to $2^{i+1} - 1$ work done on them. MLF always works on the lowest queue Q_l, and within each queue MLF processes jobs in *first come first serve* manner. If a job in Q_l has received 2^l amount of work but hasn't yet finished, it is promoted to the next queue, Q_{l+1}. Moreover, MLF will be given a constant speed-up, and we will compute this constant later.

2.2 Analysis

We define a new metric called the "inverse work" metric as an upper bound for slowdown and denote it by W^{-1}. Consider a scheduling instance J, and an algorithm A operating on it. At time step t, if $w_j(t)$ is the amount of work done on job j and $2^i - 1 \leq w_j(t) < 2^{i+1} - 1$, then define $j_t = 2^{i+1} - 1$. Then the total inverse work under A is defined as $\sum_{j \in J} \int_{r_j}^{c_j} \frac{1}{j_t}$, where r_j and c_j denote the arrival and completion times of job j respectively.

As usual, we will denote the total inverse work under algorithm A on instance J, using an s speed processor, by $W^{-1}(A, s, J)$. Similarly the optimal algorithm for W^{-1} on instance J and which uses an s speed processor will be denoted by $Opt_{W^{-1}}(s, J)$.

The main intuition of analysis is the following: Note that, any 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". A second nice 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. This two ideas form the crux of our analysis. The details of the argument are given below.

Lemma 1. *For any algorithm A, any input instance J and any $s > 0$, we have $S(A, s, J) \leq W^{-1}(A, s, J)$.*

Proof. Work done on a job is always less or equal to size of the job. Also, the job sizes are powers of 2 minus 1, we have $j_t \leq p_j$ at all times before the job is completed. Hence, the inequality follows from the definitions of slowdown and inverse work.

Given an instance J, we define $J(m)$ to be the instance on the same jobs as J, but with jobs sizes about 2^{-m} times that of J. Moreover, if the size of a job is less than 1, we consider it as 0. Formally, if $r_m(i)$ and $p_m(i)$ denote the release time and size of job i in $J(m)$, then $r_m(i) = r_i$ and $p_m(i) = \lfloor 2^{-m}(p_i + 1) \rfloor$. Let

$J' = J(1) \cup J(2) \cup \ldots \cup J(\log B)$. Thus, a job of size $2^i - 1$ in J is replaced by jobs of size $1, 2, 4, \ldots, 2^{i-1}$ in J'.

Lemma 2. *For an instance J, let J' be as defined above. Then for any $s > 0$,*

$$W^{-1}(MLF, s, J) \leq S(SJF, s, J')$$

Proof. Observe that running MLF on J exactly corresponds to running SJF on J'.

We now compare $W^{-1}(MLF, 1, J)$ with $S(SJF, 1, J')$. Let j_t be the work done by MLF on job $j \in J$ at some time t. If $j_t \leq 1$, then the contribution of j to W^{-1} is 1, and the contribution of the corresponding jobs in J' to the slowdown is at least 1. If $j_t > 1$, let $i > 0$ be such that $2^i - 1 \leq j_t < 2^{i+1} - 1$. Then the contribution of this job to $W^{-1}(MLF, s, J)$ at time t is $1/2^{i+1} - 1$. Moreover, $j_t < 2^{i+1} - 1$ means that, in J', SJF has at most finished the chunks $1, 2, \ldots, 2^{i-1}$. So the contribution of the rest of the chunks to $S(SJF, s, J')$ is at least $1/2^i$. Thus $W^{-1}(MLF, s, J) \leq S(SJF, s, J')$.

We now give a couple of useful lemmas whose proofs are fairly obvious. These will play a crucial role in our subsequent analysis.

Lemma 3. *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 $w(x, 1, j)$ denote the work done on job j, after x units of time since it arrived, under SJF using a 1 speed processor. Similarly, let $w(x, s, j)$ denote the work done on job j, after x 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, $w((t - r_j)/s, s, j) \geq w(t - r_j, 1, j)$. 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 $w((t - r_j)/s, s, j) < w(t - r_j, 1, j)$. 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 $w((t' - r_{j'}), s, j') < w(t' - r_{j'}, 1, j')$, which contradicts the minimality of t.

Lemma 4. *Let J be an instance, and $J(m)$ be defined as above. Then, if $x \geq 2^{-m}$,*

$$S(SJF, xs, J(m)) \leq \frac{1}{x} S(SJF, s, J)$$

Proof. It is easy to see that the flow time of every job $j \in J$ under SJF using an s speed processor is at least that of the corresponding job $j(m) \in J(m)$, when we run SJF on $J(m)$ using a speed $2^{-m}s$ processor.

Now, by Lemma 3, it is easy to see that if we run SJF on $J(m)$ using a speed xs processor, then the flow time of each job is $1/x$ times smaller than that if we run SJF on $J(m)$ using an $2^{-m}s$ speed processor, which is the same as that if we run SJF on J using a speed s processor.

Since the sizes of jobs in $J(m)$ are 2^{-m} times smaller than in J, the slowdown of $j(m) \in J(m)$ is $\frac{1}{x}$ times of $j \in J$.

Lemma 5. *For an instance J and J' defined as above,*

$$S(SJF, 4s, J') \leq (\log^2 B) S(SJF, s, J)$$

Proof. By Lemma 4, if we run SJF on $J(i)$ with a speed $x_i \geq 1/2^i$ processor, then we can guarantee the slowdown in $J(i)$ to be at most $1/x_i$ times the slowdown in the schedule produced by SJF on J.

So for each $J(i)$, we use a processor of speed $x_i \geq 1/2^i$, which guarantees us that slowdown on $J(i)$ is within factor $1/x_i$ of $S(SJF, 1, J)$. We run all these processors simultaneously.

Here, the total speedup needed is $\sum_{i=1}^{\log B} x_i$ and the slowdown for J' (denote by $S(J')$) is related to the slowdown of J by $S(J') \leq S(J) \sum_{i=1}^{\log B} 1/x_i$.

Choosing $x_1 = 1, x_2 = 1/2, \ldots, x_{\log\log B} = 1/\log B$, and $x_i = 1/\log B$ for $i > \log\log B$ ensures that $\sum x_i \leq 3$ and $\sum 1/x_i \leq \log^2 B$. This proves that $S(SJF, 4s, J') \leq (\log^2 B) S(SJF, s, J))$.

Theorem 1. *There is an 16-speed, $2(\log^2 B)$-competitive algorithm for minimizing mean slowdown non-clairvoyantly.*

Proof. Consider an instance J where job sizes are integers and powers of 2 minus 1. Then we have following set of inequalities:

By Lemma 1, we have $S(MLF, 8s, J) \leq W^{-1}(MLF, 8s, J)$.
By Lemma 2, we have $W^{-1}(MLF, 8s, J) \leq S(SJF, 8s, J')$.
By Lemma 5, we have $S(SJF, 8s, J') \leq (\log^2 B) S(SJF, 2s, J)$.
Lastly, by Equation 2, we have $S(SJF, 2s, J) \leq 2Opt_S(s, J)$.

Putting these together we get:

$$S(MLF, 8s, J) \leq 2(\log^2 B) Opt_S(s, J)$$

Finally, with another speed up of 2, we can get rid of the assumption that job sizes are powers of 2 minus 1(using Equation 1). Thus the result follows.

In the above, we rounded the job sizes to powers of 2, which is also the base of the logarithm in $\log B$. We use 2 as the base of the exponent in the analysis for ease of exposition. Choosing the base as $1 + \epsilon$ and a more careful analysis gives a $(1+\epsilon)$ speed, $(1+1/\epsilon)^4 \log^2 B$ competitive algorithm, for any $\epsilon > 0$. The proofs of these are deferred to the full version.

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, can be made to have at least n jobs of size 1 remaining at time $t = nB$. Thus, it incurs a tot al slowdown of at least nm. Choosing $n > B$ and $m > nB$, it is easy to see that the slowdown is at least $\Omega(B)$.

For the randomized case, we use Yao's Lemma, and observe that by time $t = nB$, the algorithm will have at least $\frac{B}{B+1}n$ 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 6. *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 $j_0, \ldots, j_{\log B}$ such that job j_i has size 2^i.

We first look at how SRPT behaves on this problem instance. The total slowdown for SRPT 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 $j_1, \ldots, j_{i-1}$ 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 $j_{i+1}, \ldots, j_{\log B}$ 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 $j_0, j_1, \ldots, j_{\log B}$. Then to finish job j_i, the scheduler has to spend at least 2^i work on at least half of $j_{i+1}, \ldots, j_{\log B}$ (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 6 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 $1, 2, \ldots 2^n$ 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, and the same is the case here. We do not need resource augmentation here, and we show that the *Round Robin* (RR) algorithm is $O(\log B)$ competitive, hence matching the lower bound shown in the previous section.

4.1 Optimal Clairvoyant Algorithm

Note that it follows from a simple interchange argument that the optimal clairvoyant algorithm for minimizing mean slowdown in the static case is exactly SRPT or equivalently SJF.

Lemma 7. *For* any *scheduling instance with* n *jobs all arriving at time* 0*, SRPT has* $\Omega(n/\log B)$ *mean slowdown.*

Proof. We approximate the total slowdown of SRPT as follows: for a job of size x, we require SRPT to work only $2^{\lfloor \lg x \rfloor}$ 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 $x_0, x_1, \ldots, x_{\log B}$ jobs of sizes $1, 2, \ldots, B$ respectively. We also have $\sum x_i = n$. Contribution to the total slowdown by jobs of size 2^i is at least $(1/2)x_i^2$. It is easy to see that $\sum (1/2)x_i^2 = \Omega(n^2/\log B)$. Thus the mean slowdown for SRPT is $\Omega(n/\log B)$.

4.2 Competitiveness of Round-Robin

The RR algorithm simply time shares the processor equally among all jobs. Thus, if there at m jobs at some time, each job gets a $1/m$ fraction of the processor.

Lemma 8. *For a scheduling instance with* n *jobs, RR has a* $O(n)$ *mean slowdown.*

Proof. Since each job shares the processor equally with at most n jobs, it can have a slowdown of at most n.

Combining the results of Lemmas 7 and 8, we get the following result:

Theorem 5. *For static scheduling with bounded job sizes, the Round Robin 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. Online 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. A. Silberschatz and P. Galvin. *Operating System Concepts*. Addison-Wesley, 5th Edition, 1998.
15. 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.