Non-Clairvoyant Scheduling for Mean Slowdown

N. Bansal [a] K. Dhamdhere [b] J. Könemann [c]
A. Sinha [d]

Dec 2001

CMU-CS-01-167

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

[b]School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. kedar@cs.cmu.edu

[c]Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA 15213. jochen@cmu.edu

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

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We consider the problem of scheduling jobs online *non-clairvoyantly*, that is, when the job sizes are not known. Our focus is on minimizing mean *slowdown*, defined as the ratio of response time to the size of the job. We first show that no (deterministic or randomized) algorithm can achieve a competitive ratio of $\Omega(n)$, where n is the number of jobs.

Resource augmentation is the concept of allowing the online algorithm a speed-up to make up for its non-clairvoyance, since the competitive ratio is obtained by comparing against an optimal *offline, clairvoyant* algorithm. We show that our lower bound continues to hold even with resource augmentation.

Finally, we consider the case when the ratio of job sizes (denoted B) is bounded. In this case, we show that any non-clairvoyant algorithm needs at least $\Omega(\log B)$ speed up to be constant competitive. We provide an algorithm which is $O(\log^2 B)$ competitive when the speed up is $O(\log B)$. In the special case when all the jobs arrive at the same time we provide an algorithm which is constant competitive and uses a $O(\log B)$ speed up.

Keywords: Scheduling, slowdown, resource augmentation, online algorithms

1 Introduction

Scheduling jobs which arrive over a period of time is a fundamental problem that occurs in several situations. A scheduler has to schedule the jobs so as to optimize a certain chosen metric, such as throughput, makespan, mean response time, or slowdown. These scheduling problems have been studied extensively in the context of approximation and online algorithms. For some nice surveys see [14, 8, 4, 10]. Most scheduling algorithms assume that the sizes of all jobs are known when they arrive. However, this need not always be the case. For example, when jobs arrive at a UNIX processor, there is often no knowledge about the processing time the jobs will require. Hence the scheduling algorithm has to perform its task *without* any clue about the sizes of the jobs. Such a scheduler is called *non-clairvoyant*. The study of non-clairvoyant scheduling algorithms was initiated by Motwani, et al [12].

Although non-clairvoyance models the reality faced by several schedulers, one may wonder what can be achieved without even seemingly basic knowledge of the job sizes. Kalyanasundaram and Pruhs [7] introduced the idea of augmenting the resources of the non-clairvoyant scheduler by increasing its speed. When the scheduler is non-clairvoyant, they proved a very remarkable result about minimizing mean response time (the response time of a job is the time it spends in the system, which can be defined as the difference between the completion time and the release time of the job). They show that if the non-clairvoyant scheduler is allowed a $(1+\epsilon)$ times faster processor, then it can achieve a response time within a $(1+\frac{1}{\epsilon})$ factor achievable by the best possible clairvoyant algorithm.

However, in certain situations average response time may not be the most appropriate measure. For example, users who are willing to pay more may expect smaller response times for their jobs. This has motivated the study of algorithms for minimizing average weighted response time [3, 9]. Another metric of interest, introduced by Bender et al [1], is the average *slowdown*, where the slowdown of a job is defined as the ratio of the response time to the size of the job. This is a special case of the weighted response time, and has been widely used to measure the "fairness" of a scheduling algorithm [11, 5, 6]. Intuitively, having a low slowdown ensures that jobs have response times in proportion to their sizes. Thus users which submit smaller jobs are equally satisfied as users which submit larger jobs.

Recently, Muthukrishnan, et al [13] considered the problem of scheduling jobs to minimize average slowdown *clairvoyantly*, that is, when job sizes are known upon arrival. They proved that the *Shortest Remaining Processing Time* (SRPT) scheduling policy is 2-competitive for minimizing mean slowdown on uniprocessor machines.

This raises the question of what can be done to minimize mean slowdown when job sizes are not known upon arrival. In this paper, we provide various lower bounds on what a non-clairvoyant algorithm can achieve for minimizing mean slowdown, for algorithms which are deterministic or randomized, and even for algorithms which have their resources augmented by means of a speed up for the processor. What is perhaps surprising here is that unlike for response times, a constant speed up given to any non-clairvoyant algorithm does not help. Even for the case when the ratio of job sizes is bounded by B, a speed up of at least $\Omega(\log B)$ is needed by any algorithm to be constant competitive.

On the positive side, we provide a non-clairvoyant algorithm which achieves competitive ratio of $O(\log^2 B)$ using a speed up of $O(\log B)$. In the special case when all jobs arrive for processing at the same time (i.e. static input), we give a non-clairvoyant algorithm which is optimal in the sense that it is constant competitive and requires only a $O(\log B)$ speed up, thus matching our lower bound.

1.1 Model

A scheduling problem of *size n* consists of a collection of n independent jobs $J = \{j_1, j_2, \ldots, j_n\}$. Each job j_i has an *execution time* or *processing requirement* (or size) p_i and a *release time* r_i. The release time of job j_i is the time at which the job is first available for processing. We assume (without loss of generality) that all job sizes are integers.

In this paper, we restrict ourselves to *preemptive scheduling* where the execution of a job can be suspended at any time and resumed later from the point of preemption without any penalty for context switching. Without pre-emption, there can be no guarantees on the performance ratio of a non-clairvoyant algorithm for minimizing mean slowdown.

A *schedule* is an assignment of some time intervals to each job on the processor, such that the total time assigned to job j_i equals p_i. No two time intervals may overlap. In this paper, we consider algorithms which are resource augmented (i.e. the processor has speed c, for some $c \geq 1$). In this case, we assume that the total time assigned by the processor to job j_i equals p_i/c.

Definition 1 *The* **Completion time** c_i *of a job* j_i *is the time at which it completes its execution. The* **response time** t_i *is the amount of time between its release and completion, i.e.* $t_i = c_i - r_i$.

Definition 2 *The* **Slowdown** *of a job* j_i *is the ratio of its response time to its size, i.e.* $s(j_i) = t_i/p_i$. *The* **slowdown** *of a set of jobs* $K \subseteq J$ *is* $\sum_{j_i \in K} s(j_i)$, *and the* **total slowdown** *of the input instance is* $S(J)$. *The* **mean slowdown** *is* $\bar{S} = S/n$.

Our goal is to minimize the mean slowdown of jobs in an input instance, where the minimum is taken over all possible input instances. Hence we are doing a worst-case analysis, as opposed to analyzing expected slowdown over a given distribution of the input.

A **clairvoyant scheduling algorithm** may use knowledge of the jobs' execution times (sizes) to assign time intervals to jobs. A **non-clairvoyant scheduling algorithm** assigns time intervals to jobs without knowing the execution times of jobs that have not yet terminated. We will always assume that the processor of the optimal algorithm has speed 1, whereas the non-clairvoyant algorithm has a speed c processor for some $c \geq 1$.

For a scheduling problem J of size n, let $\bar{S}_{A,c}(J)$ be the mean slowdown with respect to the schedule produced by algorithm A, using a c speed processor. The *optimal clairvoyant algorithm* OPT minimizes $\bar{S}_{OPT,1}(J)$ for each J. The **performance ratio** (or, in the context of online algorithms, **competitive ratio**) of a non-clairvoyant algorithm A is defined as

$$R_{A,c}(n) = \sup_{J : |J| = n} \frac{\bar{S}_{A,c}(J)}{\bar{S}_{OPT,1}(J)}.$$

A non-clairvoyant scheduling algorithm is said to be $(f(n,c), c)$**-competitive** if $R_{A,c}(n) \leq f(n,c)$. For notational convenience, we will omit c when $c = 1$.

1.2 Related work

Motwani, et al [12] first analyzed non-clairvoyant algorithms with respect to mean response time as their metric. They showed that any deterministic algorithm is $\Omega(n)$-competitive and any randomized algorithm is $\Omega(\log n)$-competitive against an optimal offline clairvoyant adversary.

More recently, Muthukrishnan, et al [13] considered the problem of online job scheduling on uniprocessor and multiprocessor machines with respect to *stretch* or *slowdown* as their metric. They show that SRPT is 2-competitive for uniprocessor and 14-competitive for multiprocessor problems respectively. However, their algorithm is clairvoyant.

Kalyanasundaram and Pruhs [7] introduced the idea of allowing a non-clairvoyant algorithm more resources than the offline algorithm. Specifically, they speed-up the non-clairvoyant processor by a factor of $(1+\epsilon)$ times the offline processor. They use this resource augmentation technique to analyze a non-clairvoyant algorithm for mean response time. They show that the competitive ratio of their algorithm is $(1+\frac{1}{\epsilon})$.

Berman and Coulston [2] improved the above results. They proved that with a v-speed processor ($v \geq 2$), the algorithm of Kalyanasundaram and Pruhs is $2/v$-competitive.

1.3 Our results

We first consider the basic problem of scheduling jobs online without knowledge of the sizes of the jobs. We show that any deterministic algorithm cannot have competitive ratio better than $\Omega(n)$ for mean slowdown. In fact, using the same instance of the scheduling problem, we show that no randomized algorithm can perform better than $\Omega(n)$. These results are proved in Section 2.1.

We next investigate (Section 2.2) the problem by applying the technique of resource augmentation. Specifically, we provide the online scheduling algorithm a processor which is k times faster than the adversary. However, we find that even resource augmentation doesn't help. We show that any deterministic or randomized algorithm is $\Omega(n/k^3, k)$ competitive.

One feature of all the examples we study to obtain these results is that job sizes keep growing in an unbounded manner. One may therefore ask what can be achieved when the ratio of job sizes is bounded. In Section 2.3, we show that if B is the ratio of job sizes, then the competitive ratio achievable by any deterministic algorithm is $\Omega(B)$.

In view of the lower bound above, we consider the case when job sizes are bounded and our non-clairvoyant processor is allowed a faster processor (Section 3 and Section 4). In this case, we show that any non-clairvoyant algorithm needs $\Omega(\log B)$ speed up to achieve a constant competitive ratio.

Our main contribution is a non-clairvoyant algorithm which is $(O\log^2 B, O(\log B))$ competitive, when the ratio of job sizes is bounded by B. In the special case when all the jobs are available at the beginning (the *static* case), we show that the *Round Robin* scheduling policy is $(O(1), O(\log B)$ competitive. Moreover, we show that this is the best possible ratio achievable in the static case.

We prove the main result by comparing the performance of our algorithm to the *Shortest Job First* (SJF) scheduling policy. Hence we also prove a theorem (in Section 5) showing that SJF is constant-competitive for mean slowdown, even when SJF is allowed to round the job sizes to powers of two.

2 Lower Bounds

2.1 Scheduling without resource augmentation

Our first result concerns non-clairvoyant scheduling for minimizing mean slowdown in the absence of resource augmentation.

Theorem 1 *Any online non-clairvoyant deterministic job scheduling algorithm has an $\Omega(n)$ performance ratio for mean slowdown.*

Proof: We present an adversary strategy which forces any non-clairvoyant algorithm A to have mean slowdown $\Omega(n)$, whereas the SRPT algorithm has a constant slowdown. This proves that A is $\Omega(n)$-competitive, since Muthukrishnan, et al [13] proved that SRPT is 2-competitive.

The adversary gives jobs in the following order. At each timestep, $i = 0, 1, \ldots, n-1$, the adversary gives 2 jobs, j_i and k_i. Let the job sets be denoted by $J = \{j_0, j_1, \ldots, j_{n-1}\}$ and $K = \{k_0, k_1, \ldots, k_{n-1}\}$. Exactly one of each j_i and k_i is of size 1. The other one is of size 2^i. Let $w_t(j)$ denote the amount of work done on job j by time t. As time proceeds, one of $w_t(j_i)$ and

$w_t(k_i)$ will reach value 1 first. Without loss of generality, call it j_i. If neither reaches 1 by time n, then let j_i be such that $w_n(j_i) \geq w_n(k_i)$. Now the adversary sets $p(j_i) = 2^i$ and $p(k_i) = 1$.

Being a work-conserving algorithm, A has done n amount of work by time $t = n$. Now $w_n(j_i) \geq w_n(k_i)$. So at least $n/2$ jobs from the set K have had less than 1 amount of work done on them. This is because if more than $n/2$ jobs in K have received one unit of work, then $w_n(k_i) + w_n(j_i) \geq 2$, and thus the total work done till time $t = n$ would be strictly greater than n, giving a contradiction.

Now among the jobs in K, let $L = \{l_1, l_2, \ldots, l_{n/2}\}$, denote the jobs which are unfinished at time $t = n$, where the order of jobs in L is the same as in K. The job l_i is in the system for at least $n/2 - i$ seconds. Thus $s(l_i) \geq n/2 - i$. Hence total slowdown for L is $S(L) \geq \sum_{i=1}^{n/2}(n/2 - i) \geq n^2/8$. Therefore the mean slowdown for jobs in L is $\bar{S}(L) = n/8$. Since there are $2n$ jobs in the entire input, we have $\bar{S} = \Omega(n)$.

On the same input instance, SRPT (which is a clairvoyant algorithm) does the following. It finishes all jobs in K (all of which have size 1) as soon as they come. So by time $t = n$, it will have finished all jobs in K, and $S(K) = n$. The jobs in J are still remaining, and will be served in order of their size. Hence the total slowdown for J's will be $S(J) = \sum_{i=0}^{n-1}(\frac{n-i+2^{i+1}-1}{2^i})$. This is clearly less than $2n + 2n$, where the first term comes from geometric progression and the second term comes from the $2^{i+1}/2^i$ part of summand.

Thus for SRPT, the total slowdown is $O(n)$. Hence the mean slowdown is $O(1)$.

This proves that any non-clairvoyant algorithm is $\Omega(n)$-competitive. □

Given that the adversary chose job sizes so as to defeat our algorithm, one might ask if randomization helps the online algorithm do better. In the next theorem, we prove otherwise.

Theorem 2 *Any randomized non-clairvoyant online scheduling algorithm for mean slowdown has $\Omega(n)$ competitive ratio.*

Proof: To prove this theorem, we use Yao's Minimax principle [15]. This states that the expected running time of the optimal deterministic algorithm for an arbitrarily chosen input distribution is a lower bound on the expected running time of the optimal randomized algorithm. Hence we only need to prove a lower bound on the expected performance ratio of any deterministic algorithm on problem instances chosen from a specific probability distribution. This gives a lower bound on the competitive ratio of all randomized algorithms.

Consider the same input instance as in Theorem 1. Let two jobs j_i, k_i arrive at time $t = i$, for $i = 0, 1, \ldots, n-1$. Now consider the following distribution on the sizes of the jobs. Independently for each i, with probability $1/2$ we have $p(j_i) = 1$ and $p(k_i) = 2^i$, and otherwise $p(j_i) = 2^i$ and $p(k_i) = 1$.

Now we look at the number N of size 1 jobs remaining in the system at time $t = n$. Let X_i denote a random variable, such that $X_i = 1$, if a size 1 job from pair i is remaining, and otherwise $X_i = 0$. Clearly, $N = \sum_{i=0}^{n-1} X_i$. Note that in a pair j_i, k_i of jobs, if $w_n(j_i) + w_n(k_i) < 1$, then $X_i = 1$. Call the number of such pairs t_0. If $1 \leq w_n(j_i) + w_n(k_i) < 2$, then at most one of j_i and k_i is finished, so $X_i = 1$ with probability at least $1/2$. Call the number of such pairs t_1. Finally, let the number of pairs for which at least 2 units of work has been done be t_2. So we have

$$\begin{aligned} t_0 + t_1 + t_2 &= n \\ t_1 + 2t_2 &\leq n \end{aligned}$$

These two equations combined imply that $t_0 + (1/2)t_1 \geq n/2$. Thus $E\{N\} = \sum_{i=0}^{n-1} E\{X_i\} \geq t_0 + (1/2)t_1 \geq n/2$.

Thus the expected number of size 1 jobs remaining at time $t = n$ is $n/2$. Again applying a similar argument as in Theorem 1, we see that expected mean slowdown for the online randomized algorithm is $\Omega(n)$. Since the input instance is the same, the clairvoyant adversary (in particular, SRPT) can again achieve a mean slowdown of $O(1)$.

This shows that for any randomized algorithm, the performance ratio is $\Omega(n)$. □

Clearly, the above bounds are achievable. Consider the *Round Robin* algorithm (RR), which works at the same rate on all jobs simultaneously. That is, for any time period $t > 0$, RR works for t/n time on each unfinished job, where n is the number of unfinished jobs. Clearly the slowdown of every job is no more than n in the Round Robin algorithm. Hence the lower bounds proved in Theorems 1 and 2 can be matched by upper bounds via the Round Robin algorithm.

2.2 Resource augmentation

Having seen that in general the competitive ratio of a non-clairvoyant algorithm can be quite bad, we now investigate the effect of resource augmentation on the competitive ratio. In resource augmentation, the online scheduler is given more resources to make up for its lack of clairvoyance. In particular, we look at the case when the online algorithm has a k times faster processor. However we find that even with a faster processor, the competitive ratio for a non-clairvoyant algorithm doesn't improve much. Formally, we prove that:

Theorem 3 *Any k-speed deterministic non-clairvoyant algorithm has $\Omega(n/k^3, k)$ performance ratio for minimizing mean slowdown.*

Proof: We modify the example from Theorem 1 to get a bad example for this case. At each timestep $t = 0, 1, \ldots, n-1$, a *group* of $k+1$ jobs arrive. Exactly one of them has size 1 (however, the non-clairvoyant algorithm doesn't know which one). The remaining k jobs have sizes $2^{ik}, 2^{ik+1}, \ldots, 2^{ik+k-1}$.

We look at how much each job has been worked on by time n. If among the ith group there was a job on which less than one unit of work was done, then the adversary chooses a job on which the smallest amount of work has been done and assigns it size 1. The rest of the sizes in that group are assigned arbitrarily. On the other hand, if our algorithm did at least 1 unit of work on all jobs in group i, then there must be a job in that group, say j_i, on which our algorithm finished doing 1 amount of work last. In that case, the adversary sets the size of j_i as 1 and thus it is finished. But at the same time, our online algorithm A was forced to do at least $k+1$ amount of work on this group.

By time n, algorithm A was able to do nk amount of work in total, since its processor operates at speed k. So if f is the number of jobs of size 1 finished, then by the above observation, algorithm A did at least $(k+1)f$ work. Hence $(k+1)f \leq nk$. Thus number of unfinished jobs of size 1 is at least $n/(k+1)$. Once again, this means that the contribution to total slowdown by these $n/(k+1)$ jobs is $\Omega(n^2/k^2)$, since the same argument as in Theorem 1 proves that each of these jobs gets a linear slowdown. Finally, since there are $n(k+1)$ jobs in all, the mean slowdown is $\Omega(n/k^3, k)$.

Once again SRPT with a speed 1 processor will have constant slowdown for this scheduling instance, again using an analysis identical to Theorem 1.

Hence the performance ratio is $\Omega(n/k^3, k)$. □

Since a k-speed processor can simulate k unit speed processors, we also have the following corollary.

Corollary 1 *Any non-clairvoyant algorithm which has its resources augmented by having k processors has an $\Omega(n/k^3)$ competitive ratio for mean slowdown.*

An argument similar to Theorem 2, using Yao's Minimax Principle allows us to prove the following.

Corollary 2 *Any k-speed randomized non-clairvoyant algorithm has an $\Omega(n/k^3, k)$ competitive ratio for minimizing mean slowdown.*

2.3 Bounded job sizes

The negative results in the previous sections all rely on unbounded job sizes. In particular, the job sizes grow exponentially with time. One might argue that this is an unrealistic assumption, and that in reality, job sizes are bounded. In this section (and in the following section), we investigate what can be done when the ratio of the maximum job size to the minimum job size is bounded by B. For simplicity, we assume that the smallest job size is 1 and the largest job size is B. Note that in this section, there is no resource augmentation.

Theorem 4 *No deterministic, non-clairvoyant algorithm has a performance ratio less than $\Omega(B)$ for mean slowdown, where B is the ratio of job sizes.*

Proof: To prove this theorem, we provide a family of input instances parametrized by a positive integer m. Our input instance is a two-stage job arrival sequence. In the first stage, mB jobs arrive at time 0. We will describe the second stage after examining the behavior of the online algorithm on the first stage jobs. If a job receives B amount of service, we declare it finished, since B is an upper bound on job sizes. Let t be the first time instant when the online algorithm gives at least $B-1$ amount of service to at least m jobs. Observe that some jobs might have finished by time t. Let x_i be the amount of service received by the jobs remaining by time t. Set the size of job i, $p_i = \min(x_i + 1, B)$.

At this point the adversary releases this information about the job sizes to the online algorithm. (This can only help the online algorithm.) Observe that there are at least $m(B-1)$ jobs left with remaining work 1 and at most m jobs with remaining work less than 1. At time t, the second stage consisting of x jobs (x will be defined later) of size 1 arrive at the rate of one per unit time for a period of x time units. We assume that the online algorithm knows that the size of the jobs that arrive after time t is 1. Once again, this is not a problem since this information can only help the algorithm.

Let us compare the behavior of the clairvoyant SRPT algorithm and the non-clairvoyant online algorithm.

Look at the $m(B-1)$ jobs with 1 unit of work remaining on them in the online algorithm. These jobs are smaller than the rest of the m jobs. So SRPT will work on the $m(B-1)$ smaller jobs first and will finish them by time t. Thus, the SRPT algorithm has at most m jobs of size B remaining at time t. Moreover, the SRPT algorithm processes the unit jobs after time t until time $t + x$ and finally finishes off the remaining jobs of size B.

On the other hand, the online algorithm at time t has at least $m(B-1)$ jobs with remaining size at least 1. Next, we observe that in order to minimize the slowdown, it is best for the online algorithm to execute the new incoming jobs of size 1.

Let S_{SRPT} and S_{ONL} denote the total slowdown of the SRPT and the online algorithms respectively. We now lower bound S_{ONL}. Observe that jobs of size 1 contribute at least x to S_{ONL}. Moreover, the $m(B-1)$ jobs remaining at time t contribute at least $\frac{xm(B-1)}{B}$ to S_{ONL}. Thus $S_{ONL} \geq xm(1 - \frac{1}{B})$.

To upper bound S_{SRPT}, observe first that the total slowdown due to the unit jobs (second stage) is x. The slowdown due to the initial $m(B-1)$ jobs which finish by time t is at most $mB \cdot mB^2$. Finally the slowdown due to the m jobs of size B which finish in the end is at most $m\frac{mB^2+x+mB}{B}$.

Thus $S_{SRPT} \leq x + m^2B^3 + m^2B + m^2 + \frac{mx}{B}$.

Setting $x = mB^4$ and letting $m > B$, we get $S_{SRPT} \leq 2m^2B^3(1 + o(1))$ and $S_{ONL} \geq m^2B^4(1 - \frac{1}{B})$, which gives the desired ratio of $\Omega(B)$ for the competitive ratio of the online algorithm. □

Note that in this example, the relation between the number of jobs n and the upper bound B on job sizes is roughly $n = O(B^4)$. Hence this lower bound is weaker than the lower bounds proved in Sections 2.1 and 2.2.

We observe from results in Sections 2.1, 2.2 and 2.3, that bounded job sizes and speed up are both necessary to obtain interesting results. The next question that arises is whether resource augmentation helps us when job sizes are bounded. We have good news on this front. We next give an algorithm which is $(O(\log^2 B), \Theta(\log B))$ competitive. While the $\Theta(\log B)$ speed up might appear excessive, we will show in Theorem 6 below that a speed of $\Omega(\log B)$ is necessary to obtain a constant competitive ratio. For the *static* case, where all jobs are available for processing at the same time, we give an $(O(1), \Theta(\log B))$ competive algorithm. We first describe the simpler static case.

3 Static scheduling

The results of Sections 2.1, 2.2 and 2.3 do not preclude a sub-linear competitive algorithm when the job sizes are bounded and we have resource augmentation. In this section we look at a restricted class of scheduling problems with bounded job sizes and resource augmentation. A scheduling problem is said to be *static* if all the release times are 0, i.e., all jobs are available for execution at the beginning itself.

Recall that the *Round Robin* algorithm (RR) works at the same rate on all unfinished jobs simultaneously. In this section, we prove that RR has a competitive ratio of $\Theta(\log B)$ for mean slowdown. Since the scheduling instance is static, this implies that RR is $(\Theta(\log B)/k, k)$ competitive. We also prove that no deterministic or randominzed non-clairvoyant algorithm is $(o(\log B/k, k)$ competitive, for the static scheduling problem. Hence, RR is optimal for static scheduling (upto constant factors).

Before we analyze RR, we note that the optimal clairvoyant algorithm for minimizing mean slowdown in the static case is exactly SRPT. This is because if the optimal algorithm ever delayed a smaller job to finish a larger job first, then the contribution to the total slowdown from these two jobs alone can be improved by making the algorithm process the smaller job before the larger job.

3.1 Competitive ratio of Round Robin

To find a bound on the competitive ratio of the Round Robin algorithm, we compare its performance on a particular input instance with that of the SRPT on the same instance.

Lemma 1 *For a scheduling instance with n jobs, RR has a $\Theta(n)$ mean slowdown.*

Proof: First we note that if a job is sharing the processor with at most k other jobs throughout its execution, then it has at most k slowdown. Also, if a job shares the processor with at least k jobs throughout its execution, then its slowdown is at least k. Since the scheduling instance has n jobs, no job can have slowdown more than n. Hence the mean slowdown is $O(n)$.

Now let us look at the execution of RR closely. Let $j_1, j_2, \ldots, j_n$ be the jobs in sorted order of size. Clearly, RR will finish the jobs in the same order. Assume, for the time being, that no two jobs have the same size. Now, the ith job must share the processor with at least $n-i$ jobs. Thus the total slowdown is at least $\sum_{i=1}^{n}(n-i) = (n-1)(n-2)/2$. Now if some $p(j_i) = p(j_{i+1})$ (that is, jobs j_i and j_{i+1} have the same size), then the slowdown for both j_i and j_{i+1} is at least $n-i$, whereas we counted it as $n-i$ and $n-i-1$. So if some jobs did have the same size, then the total slowdown is worse. Hence the total slowdown is $\Omega(n^2)$. Therefore, the mean slowdown for RR is $\Omega(n)$. □

Lemma 2 *For any scheduling instance with n jobs, the SRPT algorithm has $\Omega(n/\log B)$ mean slowdown.*

Proof: We prove this by proving a lower bound of $\Omega(n^2/\log B)$ on the total slowdown of the SRPT algorithm. As stated in the assumptions, we have integer job sizes. Let's assume that there are x_i jobs with size i, for $i = 1, 2, \ldots, B$. The total number of jobs in the instance is $\sum_{i=1}^{B} x_i = n$. With this minimal assumption, we estimate the total slowdown incurred by the SRPT algorithm.

We note that, the jobs with size i will be processed only after all the smaller jobs are finished. Thus processing of jobs with size i starts at time $\sum_{j=1}^{i-1} jx_j$. After this point, SRPT finished each of the size i job in an average $i(x_i+1)/2$. Hence, the total slowdown for the size i jobs is given by $\frac{x_i}{i}(\frac{i(x_i+1)}{2} + \sum_{i=1}^{j-1} jx_j)$. Summing this expression over all the possible job sizes, we get the total slowdown $S_{SRPT} = \sum_{i=1}^{B} \frac{x_i}{i}(\frac{i(x_i+1)}{2} + \sum_{j=1}^{i-1} jx_j)$

We divide the job sizes into $\log B$ buckets as follows: the jobs with size s, such that $2^{i-1} < s \leq 2^i$, go to bucket B_i. In the sum S_{SRPT}, we consider only the terms of the form $\frac{j}{i}x_ix_j$ (or $\frac{1}{2}x_ix_j$, if $i = j$), where i and j belong to the same bucket B_k. Also note that the x_ix_j term appears only for $j < i$. So we have $2^{k-1} < j < i \leq 2^k$. Hence $\frac{j}{i}x_ix_j \geq \frac{1}{2}x_ix_j$. Therefore, we have $S_{SRPT} \geq \sum_{k=1}^{\log B}(\sum_{i\in B_k} \frac{1}{2}x_i^2 + \sum_{j,i\in B_k, j<i} \frac{1}{2}x_ix_j$. Now, let $S_k = \sum_{i\in B_k} x_i$. This gives us: $S_{SRPT} \geq \sum_{k=1}^{\log B} \frac{1}{4}S_k^2 \geq \frac{\log B}{4}(\frac{\sum_{k=1}^{\log B} S_k}{\log B})^2 = \frac{n^2}{4\log B}$. □

Combining the results of these two lemmas, we get the following result:

Theorem 5 *For static scheduling with bounded job sizes, the Round Robin algorithm is* $O(\log B)$*-competitive.*

3.2 Lower bound

In this part, we prove a lower bound of $\Omega(\log B)$ on any deterministic as well as randomized non-clairvoyant algorithm.

Lemma 3 *No deterministic algorithm can have performance ratio better than* $\Omega(\log B)$*.*

Proof: For an arbitrary integer m, consider the following instance: m jobs of size 1, m jobs of size 2, m jobs of size 4 and so on. Since job sizes can be at most B, we have $m \log B$ jobs.

We first look at how SRPT behaves on this problem instance. Total slowdown for SRPT is $\sum_{i=1}^{B} \frac{x_i}{i}(\frac{i(x_i+1)}{2} + \sum_{j=1}^{j=i-1} jx_j)$. Here, $x_i = m$ iff i is a power of two. The first term in the summation will contribute $\frac{m(m+1)}{2}\log B$ to the total slowdown. To compute the other part, we first evaluate $\sum_{j=1}^{j=i-1} jx_j$. This is $m(2^k - 1)$ where k is the smallest power of 2 bigger than $i - 1$. Thus, $\sum_{i=1}^{B} \frac{x_i}{i}\sum_{j=1}^{j=i-1} jx_j = \sum_{2^i\leq B} \frac{x_{2^i}}{2^i} m(2^i - 1) \leq m^2 \log B$. Thus the total slowdown for SRPT is $O(m^2 \log B)$.

Now we want to show that for any non-clairvoyant deterministic algorithm A, the adversary can force the total slowdown to be $\Omega(m^2 \log^2 B)$.

The adversary adopts the following strategy: At any time t, the adversary sorts the jobs in increasing order of the amount of service received by them under the online algorithm. Let $r(J)$ denote the service received by job J at time t. Thus we have the labelling $J_0, J_1, \ldots, J_{m\log B-1}$ satisfying $r(J_0) \leq r(J_1) \leq \ldots$.

The adversary declares a job finished iff J_i receives a service of $2^{\lfloor \frac{i}{m} \rfloor}$.

Observe that this defines a valid strategy for the adversary, in the sense that the adversary can follows this strategy throughout the execution of the online algorithm and still enforce that the job sizes satisfy the conditions above.

We are now ready to prove the result. For a given online algorithm, let t_i denote the time when the first job of size 2^i finishes under the strategy adopted by the adversary.

Since the online algorithm is forced to work on at least $m(\log B - i)$ bigger jobs before time t_i, we get, $t_i > m(\log B - i)2^{i-1}$.

Thus, the total slowdown of jobs of size 2^{i-1} is at least $mt_i/2^{i-1} = m^2(\log B - i)$. Hence the total slowdown of all the jobs is at least $\sum_{i=0}^{\log B} m^2(\log B - i) = \Omega(m^2 \log^2 B)$.

Since the integer m is arbitrary, we get an infinite family of examples for which any algorithm has $\Omega(\log B)$ competitive ratio. □

One may be inclined to think that it is possible to foil the adversary's strategy by randomizing. However, we are able to show that above lower bound holds even for a randomized algorithm. We will once again use Yao's Minimax principle to prove this.

Lemma 4 *Any randomized, non-clairvoyant, online algorithm has performance ratio* $\Omega(\log B)$.

Proof: Consider the family of input instances obtained by taking all possible permutations of the input instance used in Lemma 3. We use the uniform distribution on this set of input instances. We prove a lower bound on the expected value of the total slowdown experienced by any deterministic algorithm on this input distribution.

For a given online algorithm A, let t_i denote the time when half of the jobs of size 2^i finish. We lower bound t_i as follows. Let us discount any work done by the algorithm A on jobs smaller than 2^i. We also discount the work done by A above the level of 2^i on the jobs bigger than 2^i. So, we have following simplified scenario: algorithm A does work $0 \le w_j \le 2^i$ on each job j with size $\ge 2^i$. The number of such jobs is $m(\log B - i)$. To finish half the jobs of size 2^i, A will have to do 2^i work on half of the $m(\log B - i)$ jobs (in expectation). Thus time $t_i \ge 2^i m(\log B - i)/2$. This means that, for the other half of size 2^i jobs, the slowdown is at least $m(\log B - i)$. Thus the expected slowdown for size 2^i jobs is at least $m^2(\log B - i)/2$. Hence the expected total slowdown is at least $\sum_{i=0}^{\log B} m^2(\log B - i)/2 = \Omega(m^2 \log^2 B)$.

As shown in Lemma 3, the total slowdown for SRPT is $\Theta(m^2 \log B)$. Hence the randomized algorithm has a competitive ratio of $\Omega(\log B)$. □

Theorem 6 *Any k-speed deterministic or randomized, non-clairvoyant algorithm has an* $\Omega(\log B/k)$ *competitive ratio for minimizing mean slowdown, in the static scheduling case (and hence in the online case).*

Proof: Since the input instance is a static one, a k-speed processor improves all the response times by a factor of k. Hence the mean slowdown goes down by the same factor. □

4 Dynamic Scheduling

4.1 Context

We now present an algorithm which bounds the mean slowdown when the job sizes are not known. We have seen in the earlier sections that we cannot do anything unless we have bounded job sizes, and resource augmentation. In this section, we develop an algorithm which uses resource augmentation and bounded job sizes, and has a competitive ratio of $O(\log^2 B, \log B)$ against the best possible clairvoyant scheduling algorithm, where B is the upper bound on job sizes. We begin with some preliminaries.

4.2 Algorithm Description

The input consists of a stream of jobs arriving over a period of time. More than one job may arrive at a given instant of time. Job sizes are integers. We also assume that all job sizes are powers of 2. The justification of this assumption is provided in Corollary 4.

Instead of comparing against the best possible clairvoyant scheduling algorithm, we compare ourselves against the *Shortest Job First* (SJF) policy. Formally, this policy is a pre-emptive policy which always works on the unfinished job which has the smallest size. If more than one job of the same size are available for processing, then the tie is broken according to the FCFS (First Come First Serve) rule. This implies that at any point of time, all but at most one job of any given size have received no work at all. We show in Theorem 8 that SJF is constant-competitive for minimizing mean slowdown when job sizes are powers of two.

We now describe our resource-augmented processor, and our scheduling algorithm. In view of the strong results which reinforce the difficulty of this problem, our algorithm has its resources very heavily augmented. In particular, we allow ourselves $\log B + 1$ processors, labelled $0, 1, \ldots, L = \log B$. Each of these processors runs twice as fast as the adversary's (SJF's) processor.

Our scheduling algorithm is as follows. Each processor operates on an FCFS basis. When a new job arrives, it is queued at processor 0. It becomes eligible for receiving service at processor 0 when all the other jobs preceding it disappear. At processor 0, the job receives 1 unit of work. If it is a size 1 job, then it gets finished. Otherwise it is sent to the (FCFS) queue of processor 1. For $i \geq 1$, the job receives 2^{i-1} amount of work at processor i. Thus by the time processor i gets through with the job, it has received 2^i work in all. If it does not get completed after receiving 2^{i-1} work at processor i, then it is sent to the queue of processor $i+1$. Since job sizes are bounded by B and we have $L = \log B$ processors, every job is eventually completed. We call this the *Special FCFS* algorithm, $SFCFS$ for short. Note that our policy does not need any knowledge of job sizes.

4.3 Overview of proof

When a job arrives for service in SFCFS, its slowdown is a function only of *the current state of SFCFS*, that is, its slowdown is *independent of the future*. This is easy to see from the description of our algorithm. We will critically use this idea to prove a bound on the slowdown of this job.

Our argument relies on bounding the slowdown of any newly arrived job by some portion of the slowdown experienced by SJF at the instant of time when this new job arrived. We do this bounding carefully so that different jobs charge different (non-overlapping) portions of the SJF slowdown. Hence we need to show an invariant that at any instant of time, if a job arrives, then the slowdown experienced by it is bounded by some function of the current state of SJF.

Before we state and prove the main result, we need some notation. Consider some instant of time. Let $n_i \in \mathbb{R}^+$ denote the quantity of unfinished jobs of size 2^i in SJF. That is, there are a total of $\lceil n_i \rceil$ jobs of size 2^i remaining, and one of them has already received $\lceil n_i \rceil - n_i$ amount of work. The others have not received any work, since the tie breaking rule of SJF is FCFS.

We next lower bound the slowdown experienced by a new job in SJF.

Lemma 5 *Suppose a job (J) of size 2^k arrives at time t. Let $sjf(k)$ denote the slowdown of this job under SJF. Then,*

$$sjf(k) \geq \sum_{i=0}^{k} 2^{i-k} n_i$$

Proof: By the nature of SJF, J cannot begin execution before all the work made up by jobs of size no more than 2^i is completed. (Note that we are using the FCFS tie-breaking rule of SJF here.) This quantity of work to be done is exactly $\sum_{i=0}^{k} 2^i n_i$. In fact, there may be future arrivals which further delay the execution of J. The proof follows by factoring in the size of job J, which is 2^k. $\square$

4.4 Bounding the slowdown of SFCFS: Charging

Let $s(k)$ denote the slowdown experienced by a job J of size 2^k in SFCFS if it arrives at the current instant of time. As noted earlier, $s(k)$ is not affected by future arrivals.

At this point of time, if SJF is empty, then so is SFCFS, hence J experiences slowdown 1 in both situations. Conversely, if SFCFS is not empty, then neither is SJF. We charge the slowdown of J to some portion of the slowdown experienced by some jobs which are currently alive in SJF. We use a potential function to achieve this charging. The potential function also ensures that different jobs are charged to non-overlapping portions of the slowdown of jobs in SJF.

The idea is as follows: assume there are n_i jobs of size 2^i in SJF's queue. Then, at least one of these jobs, say j_0, contributes at least n_i to the total slowdown of SJF. Whenever a job j_1 of size 2^l, $l \leq i$, arrives we will charge a part of the slowdown that SFCFS incurs for j_1 to the slowdown that SJF incurs for j_0. There are two things to take care of:

1. We must have a bound on the number of times we charge to a job j_0.
2. We are not allowed to charge to a job that SJF has finished at the current time.

In order to achieve the above two goals, we define potentials

$$0 \leq u_{ij} \leq n_i$$

for $j \leq i$. Intuitively, u_{ij} denotes the number of active jobs in SJF of size 2^i that have not yet received any charge for incoming jobs of size 2^j. Whenever a job j of size 2^l enters, we do two things:

1. We charge parts of the slowdown that j incurs in SFCFS to jobs of size 2^i for $i > l$. Whenever we charge to a job of size 2^i, we decrement u_{il}. This ensures that a single job of size 2^i does not receive too much charge.
2. We increment u_{lq} by one for all $0 \leq q \leq l$. This means that we allow smaller jobs of size 2^q for $q \leq l$ to use j's slowdown in SJF to charge their slowdown in SFCFS to.

We also want to decrement u_{ij} for all $j \leq i$ whenever SJF finishes working on a job of size 2^i. Refer to Figure 1 for a possible situation during the execution of SFCFS and SJF.

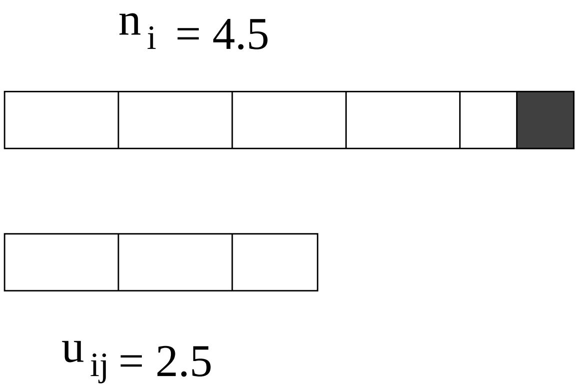

Figure 1: Relation between charge potential u_{ij} and n_i.

More formally, we let $u_{ij} = 0 = n_i \ \ \forall i, j$, initially. The following are the update rules for u_{ij}:

- **New job arrival**: On arrival of a new job of size 2^k, we increment u_{kj} for all $j = 0, 1, \ldots, k$. We don't update u_{kj} for $j > k$ since jobs of size more than k do not affect the slowdown of job J in SJF.

We also apply the following update to u_{ik} for $i > k$ and $u_{ik} > 0$:

$$u_{ik} \leftarrow \max\{u_{ik} - \frac{1}{2L}, 0\}$$

- **Passage of time**: Whenever SJF is working on a job of type i, then $\frac{du_{ik}}{dt} = -\frac{1}{2^i} \forall k \le i$ such that $u_{ik} > 0$. That is, we remove a job from the list of uncharged jobs if SJF finishes it.

The following lemma bounds the slowdown $s(k)$ that SFCFS incurs for a job of size 2^k that comes in at the current point of time. Consider the set S of jobs in the levels 0 to k, at the current time. Let $w(k)$ be the total work that needs to be done on jobs in S, so that these jobs are no longer are present in level $0, \ldots, k$. In words, jobs in SFCFS of size 2^i which are present in levels $0, \ldots, k$ contribute at most 2^k to $w(k)$, and jobs of size 2^i for $i \le k$ contribute at most 2^i.

Note that $s(k) \le w(k)2^{-k}$.

Lemma 6 *Under the update rules defined above,*

$$w(k)2^{-k} \le 2 \sum_{i=k+1}^{L} u_{ik} + \sum_{i=0}^{k} 2^{i-k} n_i$$

Before proving Lemma 6, notice that we know from Lemma 5 that the second sum on the right hand side of the above bound is at most the slowdown that SJF incurs for an incoming job of size 2^k. The first sum involves active jobs in SJF that are larger than 2^k. It is this part of the bound that we charge to larger uncharged jobs in SJF.

Proof of Lemma 6: If a job of size 2^i arrives for $i < k$, then u_{ik} is not updated. The increase in the LHS is precisely 2^{i-k}, which is identical to the increase in the RHS since n_i increases by 1.

If a job of size 2^i for $i > k$ arrives, then $w(k)$ increases by 2^k. Now the increase in u_{ik} is 2, so the inequality holds.

If a job of size exactly 2^k arrives, then both sides go up by one.

At other times, $w(k)$ is depleting by a rate of at least 2. On the other hand, SJF is either working on a job smaller than k, in which case RHS is depleting by rate 2^{-k} (because n_i goes down by 2^{i-} which is weighted by 2^{i-k}) or SJF is working on a job of size bigger than 2^k, then u_i goes down by at most 2^{-k}, thus making the rate of RHS 2×2^{-k}. Thus we have $LHS \le RHS$ at all times. □

As a corollary, we can bound the slowdown incurred under SFCFS by a job of size 2^k.

Corollary 3

$$s(k) \le 2 \sum_{i=k+1}^{L} u_{ik} + \sum_{i=0}^{k} 2^{i-k} n_i$$

Essentially, this lemma allows us to charge $s(k)$ to a portion of each of the higher size jobs in SFCFS. We charge exactly $O(\frac{1}{L})$ of $s(k)$ to the higher size jobs; hence we are able to account for only a $O(\frac{1}{L})$ fraction of the total slowdown experienced by jobs of size 2^k. Moreover, since each job in SJF creates upto L charging potentials u_{ij}'s, we find that each job in SJF is charged upto L times. Hence the total slowdown of SFCFS is no more than $O(L^2)$ times the slowdown of SJF.

This allows us to state the following theorem.

Theorem 7 *SFCFS is an $O(\log^2 B, \log B)$ competitive non-clairvoyant algorithm for minimizing mean slowdown.*

5 Shortest Job First

The scheduling policy *Shortest Job First* (SJF) always executes the shortest released job that has not yet been finished. In this section, we evaluate its performance in the dynamic, clairvoyant setting. We compare SJF against SRPT which we know is 2-competitive in this scenario. The main result is that SJF with constant speedup performs within a constant factor of SRPT on any given dynamic scheduling instance.

Our comparison of SJF and SRPT makes use of the fact that all job sizes are powers of two. We first show, that we can in fact take any given instance and transform it into an instance with job sizes that are powers of two. This does not increase the slowdown incurred by SRPT by more than a constant factor provided that we increase the speed of its processor adequately.

In the following we let SRPT_k denote the SRPT algorithm that uses a processor with speedup k.

Lemma 7 *Given a scheduling problem $J = \{j_1, \ldots, j_n\}$, release times $\{r_1, \ldots, r_n\}$ and execution times $\{p_1, \ldots, p_n\}$, define a scheduling problem $\overline{J}$ on the same jobs and with the same release times but execution times*

$$\overline{p}_i = 2^{\lceil \log p_i \rceil}$$

for all $1 \le i \le n$. We execute SRPT on instance J and let $t(SRPT, j)$ be the response time of job j. Similarly, SRPT_2 works on instance $\overline{J}$ and we denote by $t(\mathrm{SRPT}_2, j)$ the response time of job j. We then must have

$$\sum_{j=1}^{n} \frac{t(\mathrm{SRPT}_2, j)}{\overline{p}_j} \le 2 \cdot \sum_{j=1}^{k} \frac{t(SRPT, j)}{p_j}.$$

Proof: The idea is as follows: we first design a simple algorithm ALG that uses a processor with speedup two and mimics SRPT on instance $\overline{J}$. Let $t(ALG, j)$ be the time that ALG needs to finish job j. We show that $\sum_{j=1}^{n} t(ALG, j)/p_j$ is at most the slowdown of SRPT on J. We then use the fact the SRPT_2 is 2-competitive for slowdown in the dynamic, clairvoyant setting and the definition of $\overline{p}_i$ in order to bound $\sum_{j=1}^{n} t(\mathrm{SRPT}_2, j)/p_j$ in terms SRPT's slowdown.

We now define ALG: ALG works on the scheduling problem $\overline{J}$ and mimics SRPT's behavior on J. We run SRPT on instance J and let $j(SRPT, t)$ be the job executed by SRPT at time t. ALG then executes the same job $j(SRPT, t)$ at time t. If the remaining processing of $j(SRPT, t)$ for ALG is 0 at time t, we let ALG be idle.

Notice, that it follows from the definition of $\overline{p}_i$ that ALG has worked $\overline{p}_j$ time on job j at time t whenever SRPT has worked time p_j on the same job in j. In other words, the set of jobs that ALG finishes by time t is the same as the set of jobs that SRPT finishes by the same time. We immediately obtain a bound on the total slowdown of SRPT on instance J:

$$s(SRPT, J) = \sum_{j=1}^{n} \frac{t(SRPT, j)}{p_j} \ge \sum_{j=1}^{n} \frac{t(ALG, j)}{p_j}. \tag{1}$$

Recall that SRPT_2 is 2-competitive for slowdown in the dynamic, clairvoyant setting. We have

$$\sum_{j=1}^{n} \frac{t(\mathrm{SRPT}_2, j)}{\overline{p}_j} \le \sum_{j=1}^{n} \frac{t(ALG, j)}{\overline{p}_j} \le 2 \cdot \sum_{j=1}^{n} \frac{t(ALG, j)}{p_j}$$

where the first inequality follows uses the 2-competitiveness of SRPT. Combining the last inequality with (1) completes the proof. □

The following corollary to Lemma 7 justifies the assumption of exponential job-sizes.

Corollary 4 *Let J and $\overline{J}$ be defined as in Lemma 7. We now run algorithm ALG on $\overline{J}$ and let $t(ALG, j)$ be the response time of job j. Suppose, we also know that*

$$\sum_{j=1}^{n} \frac{t(ALG, j)}{\overline{p}_j} \leq c \cdot \sum_{j=1}^{n} \frac{t(\mathrm{SRPT}_2, j)}{\overline{p}_j}.$$

It then follows that

$$\sum_{j=1}^{n} \frac{t(ALG, j)}{p_j} \leq 4c \cdot \sum_{j=1}^{n} \frac{t(SRPT, j)}{p_j}.$$

Proof: It follows from $\sum_{j=1}^{n} \frac{t(ALG,j)}{\overline{p}_j} \leq c \cdot \sum_{j=1}^{n} \frac{t(\mathrm{SRPT}_2,j)}{\overline{p}_j}$ and from the definition of $\overline{p}_j$ that

$$\sum_{j=1}^{n} \frac{t(ALG, j)}{p_j} \leq 2c \cdot \sum_{j=1}^{n} \frac{t(\mathrm{SRPT}_2, j)}{\overline{p}_j}.$$

An application of Lemma 7 finishes the proof. □

Hence, from now on, we assume that all job-sizes are powers of two.

Theorem 8 *We are given a scheduling problem $J = \{j_1, \ldots, j_n\}$, release times $\{r_1, \ldots, r_n\}$ and execution times $\{p_1, \ldots, p_n\}$ such that p_i is a power of two for all $1 \leq i \leq n$. Let $s(SRPT, j)$ be the slowdown that job j incurs in SRPT's schedule. Similarly, let $s(SJF, j)$ be the slowdown that job j incurs in SJF's schedule where we assume that SJF uses a processor which is twice as fast as the one of SRPT. We also assume that SJF and SRPT break ties in the same way. Then, we must have*

$$\sum_{j=1}^{n} s(SJF, j) \leq 2 \sum_{j=1}^{n} s(SRPT, j).$$

Proof: Assume that SRPT finishes working on the given scheduling problem at time T. We say that $t \in [0, T]$ is an *event* of

[Type 1] if SRPT finishes executing a job at time t or

[Type 2] if a new job arrives at time t or

[Type 3] if SJF finishes a job at time t.

Notice, that SRPT switches active jobs only at event points of type 1 and 2. Let $e_1, \ldots, e_l$ be all events in SRPT's execution.

At any time $t \in [0, T]$, let $t(SRPT, j, t)$ ($t(SJF, j, t)$) be the total work done by SRPT (SJF) on job j. Also, for $0 \leq t \leq T$, let $j(SRPT, t)$ and $j(SJF, t)$ be the active jobs in SRPT's and SJF's schedule, respectively. If t is an event point e_i, then $j(SRPT, e_i)$ refers to the job that SRPT was working on just before event e_i, and likewise for $j(SJF, e_i)$. We maintain the following two invariants by induction over $1 \leq i \leq l$. We show that at time $t = e_i$ we have

[I1] $\forall 1 \leq j \leq n : t(SRPT, j, t) \leq t(SJF, j, t)$ and whenever $t(SRPT, j, t) \leq p_j/2$ we must have $2t(SRPT, j, t) \leq t(SJF, j, t)$ and

[I2] either $j(SRPT, t) = j(SJF, t)$, i.e. SJF and SRPT work on the same job at time t, or SJF has finished working on $j(SRPT, t)$ at time t and either

- $p_{j(SJF,t)} \geq p_{j(SRPT,t)}$ or

- $p_{j(SJF,t)} < p_{j(SRPT,t)}$ and $t(SRPT, j(SJF,t), t) = 0$, i.e. if SJF works on a smaller job than SRPT at time t then SRPT has not worked on this job at all.

Notice, that the claim trivially holds at time $e_1 = r_1$. Now, let $i > 1$ and assume that the claim holds for $1 \le j < i$. We look at three different cases depending on the type of e_i.
Case 1: e_i is of type 1
Let j be the job that SRPT just finished and let j' be the job that it switches to. We first claim that SJF must have finished job j by time e_i as well (i.e. we show that invariant I1 is maintained). It follows from our inductive hypothesis that at time e_{i-1} either SJF also worked on j or it had already finished working on j. We are fine in the latter case. So, assume that at time e_{i-1} SJF worked on job j.

Inductive hypothesis I1 tells us that

$$t(SRPT, j, e_{i-1}) \le \frac{1}{2} t(SJF, j, e_{i-1})$$

whenever $t(SJF, j, e_{i-1}) < p_j$ and

$$t(SRPT, j, e_{i-1}) \le t(SJF, j, e_{i-1})$$

otherwise. The only way SJF might have done less work on j than SRPT by time e_i is that SJF switches jobs at some time $e_{i-1} < t < e_i$. But this means that t is the arrival of a new job and hence, t must have been an event of type 2. A contradiction.

We need to take care of invariant I2. Let e_q be the last event preceding e_i where SRPT switches away from a job j'' to j such that

$$t(SRPT, j'', e_i) < p_{j''}.$$

It is clear from this definition that SRPT switches to job j'' at time e_i, i.e. $j' = j''$.

We subdivide into two cases depending on whether SJF has finished working on j' by time e_i or not.
<u>Subcase 1:</u> $t(SJF, j', e_i) < p_{j'}$
It follows by induction (I2) that SJF must have executed job j' prior to time e_q. SRPT has finished all jobs that have arrived in the interval $[e_q, e_i]$ and so has SJF (using the inductive hypothesis I1). Hence, at time e_i, j' must be the smallest job with positive remaining work for SJF as well.
<u>Subcase 2:</u> $t(SJF, j', e_i) = p_{j'}$
Assume SJF is working on job $j^\dagger$ at time e_i. We are fine if $p_{j^\dagger} \ge p_{j'}$. So, assume $p_{j^\dagger} < p_{j'}$ and hence $p_{j^\dagger} \le p_{j'}/2$. Since SRPT is switching to job j' and cannot have finished job $j^\dagger$ by time e_i, we must have

$$p_{j'} - t(SRPT, j', e_i) \le p_{j^\dagger}.$$

Let t be such that

$$p_{j'} - t(SRPT, j', t) = p_{j^\dagger}.$$

Job $j^\dagger$ must have been released after time t, i.e. $r_{j^\dagger} > t$, since otherwise SRPT would finish job $j^\dagger$ before finishing job j' which is a contradiction. But this means that $t(SRPT, j^\dagger, e_i) = 0$ since job j' was active throughout $[r_{j^\dagger}, e_i]$. This finishes the proof of case 1.
Case 2: e_i is of type 2
Assume first, that SRPT switches away from job j to the new job j' at time e_i. In this case, we must have

$$p_{j'} < p_j - t(SRPT, j, e_i)$$

and hence $p_{j'} < p_j$. It is now easy to see that SJF switches to j' too. This is clear if

$$j(SRPT, e_i) = j(SJF, e_i).$$

Suppose SJF is executing a job $j(SJF, e_i) \neq j(SRPT, e_i)$ at time e_i. In this case, the inductive hypothesis I2 guarantees that either

$$p_{j(SRPT,e_i)} \leq p_{j(SJF,e_i)}$$

or

$$p_{j(SRPT,e_i)} > p_{j(SJF,e_i)} \text{ and } t(SRPT, j(SJF, e_i), e_i) = 0.$$

We are fine in the first case. In the latter case, we can use the fact that SRPT has not worked at all on $j(SJF, e_i)$. If SRPT switches to a job $j' \neq j(SJF, e_i)$ then, we must have

$$p_{j'} \leq p_{j(SJF,e_i)}$$

and hence SJF switches to j' as well since it employs the same tie-breaking rule as SRPT.

Now, assume that SRPT does not switch to j'. Suppose that SRPT has finished less than half of job j, i.e.

$$t(SRPT, j, e_i) < \frac{p_j}{2}.$$

In this case, we know that $p_{j'} \geq p_j$. Hence, SJF does not switch either since SJF and SRPT break ties in the same way.

On the other hand suppose that SRPT has finished at least half of j. In this case, SJF has finished j using inductive hypothesis I2. Hence, assume that SJF works on another job j'' at time e_i and has switched to it at time $e_j \leq e_i$. Again, we are fine if $p_{j''} \geq p_j$. So, let's consider the case when $p_{j''} < p_j$ and hence $p_{j''} \leq p_j/2$.

Let $t \in [0, T]$ be the point in time when SRPT finishes half of job J_j, i.e.

$$t(SRPT, j, t) = \frac{p_j}{2}.$$

As before, since SRPT switches to job j' at time e_i, we must have

$$t \leq r_{j''} \leq e_i.$$

Again, by a similar argument as above, we conclude that SRPT cannot have worked on j'' at all by time e_i since j' was active throughout the whole interval.

<u>Case 3:</u> e_i is of type 3

Let j' be the job that SJF switches to at time e_i and also assume that SRPT works on job j at time e_i. We are fine whenever $p_j \leq p_{j'}$. So, assume that $p_{j'} < p_j$ and hence

$$p_{j'} \leq \frac{p_j}{2}.$$

We know that

$$p_j - t(SRPT, j, e_i) \leq p_{j'}$$

and let t be defined such that

$$p_j - t(SRPT, j, t) = p_{j'}.$$

Using the standard argument as in the preceding cases, we conclude that j' cannot have been released before time t, i.e. $r_{j'} \geq t$. But j was active throughout the interval $[t, e_i]$ and hence SRPT cannot have worked on j' at all by time e_i.

This finishes the proof since it follows that at any time $0 \leq t \leq T$ and for any job index $1 \leq j \leq n$, we must have $t(SRPT, j, t) \leq t(SJF, j, t)$. This bound on the response time immediately implies the claimed bound on the total slowdown of SRPT and SJF. □

6 Open questions

We have shown an algorithm that achieves a slowdown of $O(\log^2 B)$ with a $O(\log B)$ speed up, where B is the ratio of the size of the largest job to the size of the smallest job. We believe that it should be possible to obtain a slowdown of $O(1)$ using a speed up of $O(\log B)$ for we do not know of any stronger lower bound. It also remains open to find an algorithm which is $polylog(B)$ competitive given an $O(1)$ speed up.

Another problem which lies in between our problem and clairvoyant scheduling is that of minimizing the weighted response time where the weights of the jobs are known but the job sizes are unknown.

Acknowledgements

The authors would like to thank Avrim Blum for useful discussions.

References

[1] 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.

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

[3] C. Chekuri, S. Khanna, and A. Zhu. Algorithms for minimizing weighted flow time. In *ACM Symposium on Theory of Computing (STOC)*, pages 84–93, 2001.

[4] L. Hall. Approximation algorithms for scheduling. In *Approximation algorithms for NP-hard problems, eds. D. S. Hochbaum*, pages 1–45. PWS Publishing Company, 1997.

[5] M. Harchol-Balter, M. Crovella, and C. Murta. Task assignment in a distributed server. In *Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, 1998.

[6] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley, New York, 1991.

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

[8] D. Karger, C. Stein, and J. Wein. Scheduling algorithms. In *CRC handbook of theoretical computer science*, 1999.

[9] H. Kellerer, T. Tautenhahn, and G. J. Woeginger. Approximability and nonapproximability results for minimizing total flow time on a single machine. In *ACM Symposium on Theory of Computing (STOC)*, pages 418–426, 1996.

[10] E. Lawler, J. Lenstra, A. Kan, and D. Shmoys. Sequencing and scheduling: algorithms and complexity. In *Logistics of Production and Inventory: Handbooks in Operations Research and Management Science*, volume 4, pages 445–522. North-Holland, 1993.

[11] M. Mehta and D. J. DeWitt. Dynamic memory allocation for multiple-query workloads. In *International Conference on Very Large Data Bases*, pages 354–367, 1993.

[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] J. Sgall. Online scheduling. In *Online Algorithms: The State of the Art, eds. A. Fiat and G. J. Woeginger*, pages 196–231. Springer-Verlag, 1998.

[15] A. C-C. Yao. Probabilistic computations: Toward a unified measure of complexity (extended abstract). In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1977.