.

Algorithms for Flow Time Scheduling

Nikhil Bansal

December 2003

CMU-CS-03-209

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Avrim Blum, Chair
Bruce Maggs
Kirk Pruhs, University of Pittsburgh
R. Ravi

Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

This research was sponsored by the National Science Foundation (NSF) under grant nos.CCR-0105488, CCR-0122581 and CCR-9902091, by the Pittsburgh Digital Greenhouse (PDG) and the IBM Corporation through a generous graduate fellowship. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the NSF, the PDG, IBM, the U.S. government or any other entity.

.

Keywords: Approximation Algorithms, On-line Algorithms, Scheduling, Flow Time, Non-clairvoyant Scheduling

Abstract

We study scheduling algorithms for problems arising in client-server systems. In the client-server setting, there are multiple clients that submit requests for service to the server(s) over time. Typical examples of such systems include operating systems, web-servers and database query servers. As there could be multiple clients requesting a service, the goal of a scheduling algorithm is to provide service to the clients in some reasonable way. A natural measure of the quality of service received by a client is its flow time, defined as the time since the client submits a request until it is completed. In this thesis, we study some fundamental problems related to minimizing flow time and its variants. These include ℓ_p norms of flow time, weighted flow time, stretch (flow time divided by its service requirement) and completion time. We consider these problems in various settings, such as online, offline, scheduling when the processing requirements are unknown and scheduling when jobs can be rejected at some cost.

Acknowledgments

First of all, I would like to thank my advisor Avrim Blum for his infinite guidance, support and insights. I will be indebted to him for all the ways in which I have grown as a researcher and especially for teaching me how to think. Clearly, this thesis would not be possible without him.

I have been really fortunate to have Kirk Pruhs at such a close distance. He has been like a second advisor to me. I am extremely grateful to him for sharing his invaluable ideas and his limitless enthusiasm for research. A large part of this thesis has its origins in the discussions at the Pitt lunch trucks.

I want to thank all the theory folks at CMU for providing such a wonderful and stimulating research environment. Special thanks to Avrim Blum, Tom Bohman, Alan Frieze, Anupam Gupta, Ryan Martin, R. Ravi and Steven Rudich for their extraordinary classes.

Thanks to Shuchi Chawla, Kedar Dhamdhere, Amitabh Sinha, Jochen Konemann and Adam Wierman for being such great friends, colleagues and collaborators. Many thanks to Ashwin Bharambe, Amit Manjhi, Mahim Mishra and Bianca Schroeder for all the nice time I spent with them.

Finally, I would like to thank my parents for their unconditional love and encouragement and instilling in me the love of learning. A special thanks to my friend Ageeth for all her love and understanding.

Contents

Chapter 1

Introduction

Scheduling can be described as allocating resources over time. The resources are modeled as machines and the requests for resources are modeled as jobs with a service requirement. Given this broad definition, scheduling has been studied in various scenarios such as practical computer systems, production planning in factories, load balancing and so on. Scheduling problems constitute a substantial part of optimization, and have been studied by researchers in various communities such as operations research, algorithms and queueing theory.

In the last ten years or so, a major focus has been to study the performance of scheduling algorithms for client-server systems. In a client-server system, there are multiple clients that submit requests for service to the server(s) over time. Typical examples of such systems include operating systems, web-servers and database query servers. In such systems, the most natural measure of the quality of service received by a client is the *flow time*, defined as the time since the client submits a request until it is completed. Flow time closely relates to the user experience as it measures the amount a user has to wait to get his jobs serviced. In this thesis we look at some fundamental problems related to minimizing flow time and its variants.

1.1 Motivation

Our central motivating question is: *What is a good scheduling algorithm for a client-server system?* There is no single answer to this question. The answer would depend on the particular type of scenario, such as

- Is a single machine or multiple machines used to serve the requests? For example, a CPU may consist of a single processor or multiple processors. Similarly, a web site might use several servers to serve its content instead of a single one.

- Is the processing requirement of a job known upon its arrival? For example, in a web server serving static documents, the time to serve a request is reasonably modeled by the size of the file which is known to the server. On the other hand, an operating system typically has no idea of long it would take to serve a request.

- Can the execution of a job be interrupted arbitrarily and resumed later from the point of interruption without any penalty? There might be a significant overhead in some systems to switch between jobs.

- Are all jobs equally important? It might be important to give some jobs preferential treatment relative to other jobs.

However, in spite of the specifics of the model, one can usually give some general guidelines.

First, one would like that the average time that users experience to finish their jobs is not too high. One way to formalize this goal is to minimize the total flow time. Similarly, one might wish that most users receive similar performance, that is, it should not be the case that one set of users receives significantly worse performance than others. This can be formalized by requiring that maximum flow time be minimized. Optimum algorithms are known both for minimizing the total flow time and for minimizing the maximum flow time. However, these algorithms are totally "incompatible" with each other. The algorithm for minimizing the total flow time might lead to extremely high maximum flow time and similarly the algorithm for minimizing the maximum flow time might lead to extremely high total flow time. Our first motivating problem is to study algorithms which perform reasonably well on both the measures simultaneously. We formalize this goal by introducing the problem of trying to minimize the ℓ_p norms of flow time. We study this problem is different settings. In particular, whether the service requirement of a job is known upon its arrival or not, whether all jobs are equally important or not and finally where is goal is to minimize the ℓ_p norms of a related measure known as stretch (defined as the ratio of a job's flow time to its service requirement).

Often in the real world, not all jobs are equally important. For example, some users might pay a higher amount of money to their Internet service providers to receive a better quality of service, or a website serving some content might have different quality of service guarantees for different classes of users and so on. Sometimes such priorities might simply be inherent in the system. For example in operating systems, the basic kernel operations receive the highest priority and similarly the I/O related jobs receive a higher priority than jobs involving a lot of processing. A standard way in the scheduling literature to formalize the notion of varying degrees of importance is to associate a weight with a job which is indicative of its priority. The goal then is to optimize some weighted measure of the flow times. Unfortunately, scheduling in the weighted scenario is much less understood than in the unweighted scenario. A fundamental problem that is not well understood is that of minimizing the total weighted flow time on a single machine. Prior to our work, no "truly" online algorithm was known for the problem. Moreover no algorithm with an $O(1)$ approximation ratio was known even when there are just two different weights. In this thesis, we give an $O(1)$ approximation for the case where there are $O(1)$ weights, and more generally an $O(\log W)$ competitive algorithm where W is the ratio of the maximum to the minimum weight.

A third fundamental and open problem is understanding the approximability of the problem of minimizing the total flow time (unweighted) on multiple machines. Even for two machines, the best known result is an $O(\log n)$ approximation algorithm, where n is the number of jobs. On the other hand, it might be possible that there is a polynomial time approximation scheme even for the case of arbitrary number of machines. In this thesis, we give a quasi-polynomial time approximation scheme for the problem when there are a constant number of machines. This suggests that a polynomial time approximation scheme is likely to exist for this case.

In this thesis, we describe our progress on the problems mentioned above and some other related questions. A detailed summary of our work is given in Section 1.5. We begin by describing the model formally.

1.2 Model and Preliminaries

An instance of a scheduling problem consists of a collection of jobs $\mathcal{J} = \{J_1, \ldots, J_n\}$ which arrive dynamically over time. The time when a job J_j arrives is its *release time* and is denoted by r_j. Each job has a service requirement, also known as its *size* and is denoted by p_j. We will also consider problems where the jobs are weighted according to their importance, in this case the weight of a job J_i is denoted by w_i. B

will always denote the ratio of the maximum to minimum job size and W will always denote the ratio of the maximum to minimum weight. The number of jobs is always denoted by n, and the number of machines is always m. A *schedule* for a problem instance specifies, for each job, various intervals of time where this job is executed. Figure 1.1 shows a possible schedule for $n = 3$ jobs and $m = 1$ machine. A scheduling algorithm is a procedure to construct a schedule from an input instance.

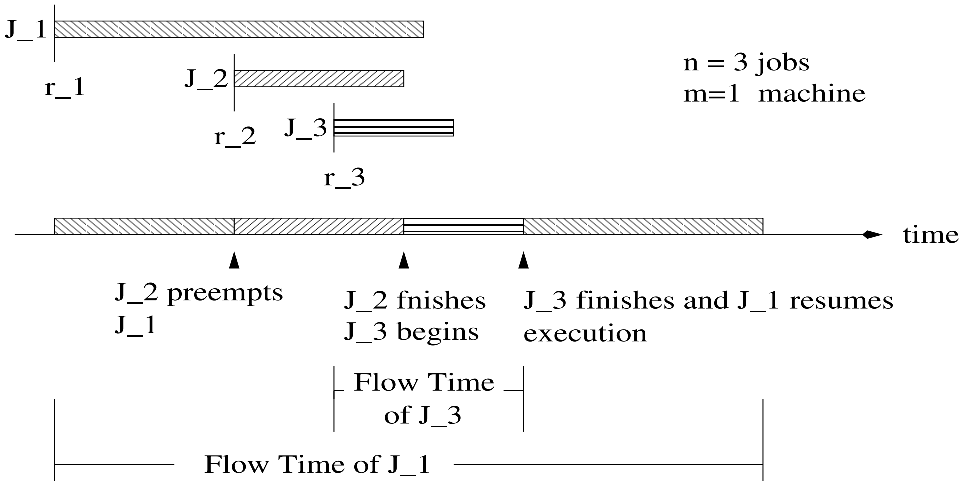


Figure 1.1: An example of a schedule

Typically scheduling problems are studied in three paradigms based on how the input is revealed.

Offline: In this paradigm, the entire input instance is known in advance and the goal usually is to give an efficient (polynomial time) algorithm that produces the optimum or *close to* optimum schedule.

Online: In this paradigm, the existence of a job is revealed to the algorithm only when it arrives (i.e. J_i becomes known at time r_i). Thus the *online algorithm* at any time has to make decisions based on the partial knowledge of the jobs that have arrived by that time. This paradigm is perhaps the most natural and appealing one in the context of scheduling as far as real world applications are concerned. The online paradigm is further classified into two paradigms based on what information is revealed about a job when it arrives.

- *Online Clairvoyant Scheduling:* In this paradigm, the size p_i of a job J_i is revealed to the algorithm when it arrives at time r_i. This models well for example the scenario in a web-server serving static documents. Typically, the time to serve a request can be reasonably modeled by the size of the file requested, which is known to the web-server.

- *Online Non-clairvoyant Scheduling:* In this paradigm, only the existence of the job J_i is revealed to the algorithm at time r_i, and in particular the size of the job is unknown. The algorithm learns the size of the job only when it meets its service requirement and terminates. This paradigm was first introduced formally and studied by Motwani, Philips and Torng [59]. It closely models the scenario faced for example by operating systems, since the system typically will have no idea of the amount of processing required by a job.

Objective Functions

Given a schedule, the completion time c_j for a job is the time at which it completes its service requirement. The flow time f_j of a job is time a job spends in the system. i.e. it is the difference between the completion time and the release time ($f_j = c_j - r_j$). Figure 1.1 shows the flow times of jobs J_1 and J_3. The term flow time unfortunately is not a standard one, often flow time is also referred to as response time, sojourn time, latency or waiting time in the literature.

Another measure closely related to flow time is *stretch*. Given a schedule, the stretch s_j of a job is the ratio of its flow time and its size i.e. $s_j = f_j/p_j$. Stretch can be thought of as *normalized* flow time, and relates the waiting time of the users to their demands. In particular, it models the fact that users are usually willing to wait longer for large jobs as opposed to short jobs. Again, stretch is sometimes referred to as slowdown or normalized flow time in the literature.

Preemption

A schedule is *non-preemptive* if a job cannot be interrupted once it is started. In a *preemptive* schedule a job can be interrupted arbitrarily and the execution can be resumed from the point of interruption without any penalty (for example in Figure 1.1, the job J_2 preempts J_1).

Being unable to preempt places a serious restriction on the quality of schedules with respect to the flow time and stretch metrics. For online algorithms in the non-preemptive setting, one can usually construct a simple example showing that any algorithm will be arbitrarily bad. Similarly, for approximation algorithms one can usually construct a hard instance based on a standard construction due to Kellerer, Tautenhahn and Woeginger [48], who use it to show an inapproximability result of $\Omega(n^{1/3})$ for minimizing total flow time non-preemptively on a single machine. Thus, flow time and stretch related objectives functions are almost always studied in the preemptive model. In the thesis, we focus only on preemptive scheduling.

1.3 Analysis Framework

Throughout this thesis, we will use the standard worst case analysis.

1.3.1 Offline Analysis

For offline problems, the goal is to give an efficient (polynomial time) procedure to construct an optimum solution. However, often the problems that arise are $\mathcal{NP}$-Hard and it is unlikely that optimum solutions for these can be computed exactly in polynomial time. The goal then is to construct an algorithm which is efficient and computes a solution that is provably close to optimum for every instance I. Formally,

A (deterministic)[1] algorithm Alg for a minimization problem Π is called a *ρ-approximation* algorithm if

$$Alg(I) \leq \rho Opt(I)$$

holds for every instance I of Π, where $Alg(I)$ denotes the cost of Alg on I and $Opt(I)$ denotes the optimum cost on I. The number ρ is called the *approximation ratio* of the algorithm Alg. Usually one requires an

[1] One can also define randomized approximation algorithms, however in this thesis we will only consider deterministic approximation algorithms.

approximation algorithm to run in time polynomial in the input parameters such as the number of jobs. A nice introduction to approximation algorithms can found in [37, 74].

1.3.2 Online Analysis

We use the standard technique of competitive analysis introduced in the seminal paper of Sleator and Tarjan [71]. An online algorithm Alg is called *c-competitive* if the objective function value of the solution produced by Alg on any input sequence is at most c times that of the optimum offline algorithm on the same input. Here, the optimum offline algorithm has complete knowledge about the whole input sequence in advance. Formally,

A deterministic[2] online algorithm is called c-competitive if

$$Alg(I) \leq c\, Opt(I)$$

holds for every instance I. The number c is called the competitive ratio of the algorithm Alg. Sometimes, the definition is relaxed to allow a constant b (independent of the input sequence) such that

$$Alg(I) \leq c\, Opt(I) + b$$

Observe that there is no restriction on the computational resources of an online algorithm. Competitive analysis of online algorithms can be imagined as a game between an online player and a malicious adversary. If the adversary knows the strategy of the online player, he can construct a request sequence that maximizes the ratio between the player's cost and the optimum offline cost. A nice and in-depth treatment on online algorithms and competitive analysis can be found in [19].

In the non-clairvoyant setting, the performance is measured in a similar way. In particular, an algorithm is c-competitive if the objective function value of the solution produced by Alg on any input sequence is at most c times that of the optimum offline algorithm (and hence clairvoyant) on the same input.

1.3.3 Resource Augmentation

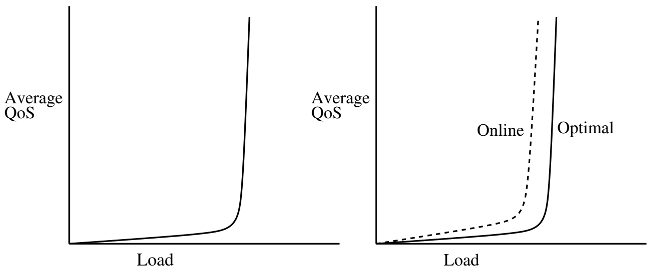
Sometimes competitive analysis turns out be overly pessimistic and the adversary is simply too powerful and allows only trivial competitiveness results. An alternative form of analysis that has proven very useful in the context of scheduling is *resource augmentation*, introduced by Kalyanasundaram and Pruhs [46]. The idea here is to augment the online algorithm with extra resources in the form of faster processors.

Formally, let Alg_s denote an algorithm that works with processors of speed $s > 1$. An online algorithm is called $s-$speed, $c-$competitive if

$$Alg_s(I) \leq c\, Opt_1(I)$$

holds for all input instances I. Another way to view resource augmentation is that while we would like our algorithm to perform almost as well as the optimum, we may be unable to do so if we allow the optimum the same amount of resources. Thus, we compare the performance of our algorithm to that of the optimum where the optimum is only allowed an s times slower processor. The typical goal in the resource augmentation setting is to find an algorithm which is $O(1)$ competitive with a modest amount of speed up. To understand this goal, we first need to understand how client-server systems typically behave.

[2]Again, one can define randomized online algorithms, and various notions of an *adversary* based on the kind of information about the online algorithm available to it. However, we will only focus on deterministic online algorithms.



(a) **(b)**

Figure 1.2: (a) Standard QoS curve, and (b) The worst possible QoS curve of an $(1 + \epsilon)$-speed $O(1)$-competitive online algorithm.

Average QoS curves such as those in figure 1.2(a) are ubiquitous in server systems [51]. That is, there is a relatively modest degradation in average QoS as the load increases until some threshold is reached — this threshold is essentially the capacity of the system — after which any increase in the load precipitously degrades the average QoS. The concept of load is not so easy to formally define, but generally reflects the number of users of the system. If A is an $(1 + \epsilon)$-speed, c-competitive server scheduling algorithm, and $\textit{Opt}_1(I) \leq d \cdot \textit{Opt}_{1+\epsilon}(I)$ then A is at worst $(c \cdot d)$-competitive on input I. The loads in the usual performance curve shown in figure 1.2(a) where $\textit{Opt}_1(I)$ is not approximately $\textit{Opt}_{1+\epsilon}(I)$ are those points near or above the the capacity of the system. Thus the performance curve of a $(1 + \epsilon)$-speed, c-competitive online algorithm A should be no worse than shown in figure 1.2(b). That is, A should scale reasonably well up to quite near the capacity of the system. Thus an ideal resource augmentation result would be to prove an algorithm is $(1 + \epsilon)$-speed, $O(1)$-competitive. More detailed discussion on resource augmentation and its uses can be found in [46, 61, 63].

1.4 Preliminaries

A local view of Flow Time

Often while studying the performance of an algorithm Alg for some flow time related measure, it is helpful to think of how the measure changes as a function of time. This usually has a nice physical description. For example, at any time the rate of change of total flow time under Alg is simply the number of unfinished jobs under Alg at that time. Similarly for weighted flow time $\sum_j w_j f_j$, it is easily seen that the rate of change is equal to the total weight of unfinished jobs at that time.

This leads to a simple and often useful technique known as *local-competitiveness*. The idea is that, to prove c-competitiveness of Alg it suffices to show that at all times t, the rate at which Alg increases is no more than c times the rate at which any algorithm Opt can increase at that time. For example, to prove that an algorithm is c competitive for weighted flow time, it would suffice to show that at any time the total weight of unfinished jobs under the proposed algorithm never exceeds c times the weight under any schedule at that time.

The advantage of local-competitiveness is that it is often easier to show some local property of an algorithm than to argue against globally optimum schedules. Of course, local-competitiveness is not always applicable, it could be the case that no matter how good the online algorithm is globally, there are short periods during which the adversary can perform significantly better.

Typical Scheduling Algorithms

Most often in scheduling, there are only a few algorithms which end up being analyzed for different problems. We now describe these and some associated basic results.

1. First In First Out (*FIFO*): At any time, work on the job which arrived the earliest.

   It is a folklore result that *FIFO* is optimal for minimizing the maximum flow time on a single machine.

2. Shortest Job First (*SJF*): At any time, work on the job that has the shortest size (p_i).

3. Shortest Remaining Time First (*SRPT*): At any time, work on the job that has the least remaining processing requirement at that time.

   It is a folklore result that *SRPT* is optimal for minimizing the total flow time on a single machine. Recently, it was shown that *SRPT* is 2-competitive for minimizing total stretch on a single machine [60].

4. Shortest Elapsed Time First (*SETF*): At any time, work on the job that has received the least amount of service at that time. Note that by the nature of the algorithm, at any time if two of more jobs have the least amount of service, then they continue to have the least amount of service and share the processor equally until a new jobs arrives.

   A celebrated result of Kalyanasundaram and Pruhs is that *SETF* is $(1+\epsilon)$-speed, $(1+1/\epsilon)$-competitive for minimizing total flow time non-clairvoyantly on a single machine [46].

5. Round Robin (*RR*): At any time, share the processor equally among all the jobs present at that time. This policy is also referred to as *Processor Sharing* in the literature.

Algorithms when jobs have weights:

1. Highest Density First (*HDF*): At any time work on the job that has the largest weight to processing time ratio. The ties are broken in favor of the partially executed job.

   It is known that *HDF* is a $(1+\epsilon)$-speed, $(1+1/\epsilon)$-competitive algorithm for minimizing the total weighted flow time [16].

2. Weighted Shortest Elapsed Time First (*WSETF*): For a job J_i with weight w_i , let $p_i(t)$ denote the amount of work done on J_i by time t. We define the normalized service of J_i as $\|J_i\|_t = p_i(t)/w_i$. At all times, *WSETF* splits the processor among the jobs that have the smallest normalized service in the ratio of their weights. For example, if $J_1, \dots, J_k$ are the jobs that have the smallest normalized service. Then J_j, for $i = 1, \dots, k$, will receive $w_j/(\sum_{i=1}^{k} w_i)$ fraction of the processor. Note that for all jobs J_i that *WSETF* executes, the normalized service increases at the same rate and thus stays the same.

   One of the results of this thesis is that *WSETF* is $(1+\epsilon)$-speed, $(1+1/\epsilon)$-competitive for minimizing the total weighted flow time non-clairvoyantly [7].

Non-Clairvoyant Scheduling

Finally, to place our results in the proper perspective we give a quick (and incomplete) history of non-clairvoyant scheduling. Non-clairvoyant scheduling was first introduced in the context of online algorithms by Motwani, Philips and Torng [59]. They studied the problem of minimizing the total flow time on a single machine and showed that any deterministic algorithm is $\Omega(n^{1/3})$ competitive and that any randomized algorithm is $\Omega(\log n)$ competitive. On the positive side, Kalyanasundaram and Pruhs gave a randomized algorithm with an almost matching competitive ratio of $O(\log n \log\log n)$ [45]. Later, Becchetti and Leonardi improved the analysis of [45] to show that their algorithm in fact achieves a competitive ratio of $O(\log n)$ [13]. Resource augmentation was also first introduced to deal with the problem of non-clairvoyance and was used to show that *SETF* is $(1+\epsilon)$-speed, $(1+1/\epsilon)$-competitive [46].

1.5 Overview of Results

We now outline the problems considered in this thesis.

Minimizing the ℓ_p Norms of Flow Time

It is well known that *SRPT* is optimal for minimizing the *total* flow time. Similarly, *FIFO* is optimal of minimizing the *maximum* flow time. However, each of the algorithms has its drawbacks. *FIFO* might have a very poor average behavior (for example, consider $n-1$ small jobs stuck behind a large job). Similarly, a commonly cited reason against using *SRPT* in a real system is the fear that it may "starve" large jobs (i.e. even though the average performance is good, there might be a few large jobs with extremely high flow times). To strike a compromise between the sum and the maximum, we propose the ℓ_p norms of flow time as a measure of good average performance and fairness. While ℓ_p norms of measures such as completion time and load have been considered previously [30, 3, 5], no previous work has been done on flow time.

To get a feel for the problem, let us consider the case when $p = 2$. In this case, at any time, the contribution of an unfinished job to the flow time squared is equal to two times its *age* (which is defined as the time spent by the job thus far). Thus, at any time, the total flow time squared increases at the rate of the total age of all the unfinished jobs at that time. This suggests that any reasonable algorithm for the problem must try to avoid a lot of "old" jobs from building up.

We obtain the following results:

- We first show that there *cannot* be any randomized online algorithm with a competitive ratio of $n^{o(1)}$ for $1 < p < \infty$. This is perhaps surprising as it may be tempting to believe that a suitable combination of *SRPT* (keeps total number of jobs low) and *FIFO* (gets rid of old jobs) should yield a good online algorithm.

- Motivated by the lower bounds above, we consider the problem in the resource augmentation model. We show that *SJF* and *SRPT* are $(1+\epsilon)$-speed, $O(1)$-competitive algorithms for minimizing all ℓ_p norms of flow time.

  Our proofs for these results use local competitiveness. We show that with a slight speed up, *SJF* and *SRPT* not only keep the total number of jobs low, but also do not allow a buildup of old jobs. The proof requires a careful accounting of the number of jobs of various ages at all times.

Algorithm	Speed	Competitive Ratio
Any Clairvoyant Algorithm	1	$n^{\Omega(1)}$ for $1 < p < \infty$
SJF	$(1+\epsilon)$	$O(1/\epsilon)$
SRPT	$(1+\epsilon)$	$O(1/\epsilon)$
SETF	$(1+\epsilon)$	$O(1/\epsilon^{2+\frac{2}{p}})$
RR	$(1+\epsilon)$	$\Omega(n^{(1-2\epsilon p)/p})$
Any Non- clairvoyant Algorithm	$(1+\epsilon)$	-

Table 1.1: Results for ℓ_p norms of Flow Time

Somewhat surprisingly, we also show that the policy *RR* which is aimed at giving a fair performance to all jobs, does not have the above property. In particular, we show that even with a $(1+\epsilon)$ speed up (for ϵ sufficiently small), *RR* has a competitive ratio of $\Omega(n^{(1-2\epsilon p)/p})$.

- Finally, we consider this problem in a non-clairvoyant setting. We show that *SETF* is a $(1+\epsilon)$-speed, $O(1)$-competitive algorithm.

  It can be shown that local-competitiveness does not yield useful results in this setting. To prove this result for *SETF* we introduce a technique where, by a series of transformations we reduce the problem of proving the result for *SETF* to a result about *SJF*. This technique is fairly general and turns out to be useful to prove other results about non-clairvoyant algorithms. For example, we will use this later to show non-trivial results about minimizing the total stretch and the ℓ_p norms of stretch non-clairvoyantly.

These results are summarized in Table 1.1 below.

Our results argue that the concern, that the standard algorithms aimed at optimizing average QoS might unnecessarily starve jobs, is unfounded when the server is less than fully loaded. Results similar in spirit are found in a series of papers, including [9, 27, 35, 65]. These papers argue that *SRPT* will not unnecessarily starve jobs any more than Processor Sharing does under "normal" situations. In these papers, "normal" is defined as there being a Poisson distribution on release times, and processing times being independent samples from a heavily tailed distribution. More precisely, these papers argue that every job should prefer *SRPT* to Round Robin under these circumstances. So informally our results and these papers reach the same conclusion about the superiority of *SRPT*. But in a formal sense the results are incomparable.

Minimizing Weighted Flow Time

In many settings jobs have varying degrees of importance and this is usually represented by assigning weights to jobs. A natural problem that arises in this setting is to minimize the total weighted flow time. This problem is known to be $\mathcal{NP}$-Hard [53].

Since *SRPT* at any time has the minimum possible number of jobs in the system, it easily follows that *SRPT* is an $O(W)$ competitive algorithm for weighted flow time. Similarly, the policy that works on the highest weight job at any time is easily seen to be an $O(B)$ competitive algorithm. These were essentially the best known guarantees for the problem until the recent work of Chekuri, Khanna and Zhu [23], who gave an $O(\log^2 B)$ competitive semi-online[3] algorithm. There also show a lower bound of 1.61 on the competitive ratio of any online algorithm. However, their work left some questions open. In particular, is there an $O(1)$ competitive algorithm for minimizing weighted flow time with arbitrary weights? A simpler problem left open was whether there is an $O(1)$ competitive algorithm even if there are only two different job weights? Finally, they also ask if a there is "truly" online algorithm with a non-trivial competitive ratio.

In our work, we answer some of these questions. We give a "truly online" algorithm that is k competitive if there are k different job weights. Thus, this gives the first algorithm that is $O(1)$ competitive for 2 different job weights. Rounding the weights to powers of 2, it is easy to see that our algorithm gives an $O(\log W)$ competitive algorithm for arbitrary weights.

Our algorithm is quite simple. We divide the jobs into $O(\log W)$ classes. At any time, the algorithm works on the weight class with the greatest weight of unfinished jobs. Within a class jobs are processed in the ***SRPT*** order. Thus, in a sense it tries to keep the total weight in each class low. However, the crucial point to show is that the algorithm does not end up performing badly as it moves between various classes. Our proof uses local competitiveness: We show that at any time, the total weight of unfinished jobs in our algorithm is $O(\log W)$ times that under any other algorithm.

While the above algorithm has a guarantee in terms of W, using a standard trick, we show how this algorithm can be modified to give an $O(\log n + \log B)$ semi-online algorithm for the problem.

Interestingly, our results described above and those of [23] are also the best known guarantees for the offline version of the problem. On the other hand, we do not even know if the problem is $\mathcal{APX}$-Hard. Though a recent result of Chekuri and Khanna gives a quasi-polynomial time approximation scheme for this problem [21], which suggests that a polynomial time approximation scheme is likely to exist for this problem.

We next consider an extension of the problem where we seek to minimize the weighted ℓ_p norms of flow time, defined as $(\sum_i w_i f_i^p)^{1/p}$. We study this problem both in the clairvoyant and non-clairvoyant setting. From the lower bounds on minimizing the ℓ_p norms of *unweighted* flow time (mentioned in Table 1.1), it easily follows that there do not exist any $n^{o(1)}$ competitive clairvoyant (and hence non-clairvoyant) algorithms for this problem, where $1 < p < \infty$.

We show that ***HDF*** and ***WSETF***, the natural generalizations of ***SJF*** and ***SETF*** to the weighted case are $(1 + \epsilon)$-speed, $O(1)$-competitive for minimizing the weighted ℓ_p norms of flow time. To prove these results, we give a general technique which reduces a weighted problem to an unweighted one, but requires an additional $(1 + \epsilon)$ speed up in the process.

Stretch Scheduling

Stretch as a measure of system performance has gained a lot of popularity in recent years. Stretch is very appealing as it directly quantifies the time spent waiting by a user per unit amount of received service. Often computer system designers consider stretch as a measure while demonstrating the superiority of their system over others. Theoretically, stretch is reasonably well understood, at least in the clairvoyant setting. It is known that ***SRPT*** is 2-competitive for minimizing the total stretch on a single machine [60]. On multiple machines, $O(1)$ competitive algorithms for minimizing total stretch in various settings [60, 23] are known. In the offline setting, a PTAS is known for minimizing total stretch on a single machine [18, 21].

In this thesis we study this problem in the non-clairvoyant setting. The problem was posed in [13], and

[3] Their algorithm is semi-online as it requires the knowledge of B while scheduling the jobs.

arises naturally is systems such as Operating systems, where the jobs sizes are unknown and where stretch is considered to an important measures of user satisfaction.

Minimizing total stretch 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). Formally, this hardness is seen by the fact that any randomized algorithm is $\Omega(\min(n, B))$ competitive for this problem.

To get a feel for minimizing stretch non-clairvoyantly, suppose there are n jobs of size $1, 2, 4, \ldots, 2^{n-1}$ all of which are released at time $t = 0$. By scheduling these in the order of increasing job sizes, the optimal clairvoyant algorithm has a total stretch of $O(n)$. However, the non-clairvoyant has no way of figuring out the size 1 job until it has spent at least 1 unit of time on all the size 1 jobs (or on about half the jobs if the algorithm is randomized). Thus the size 1 job will have a stretch of $\Omega(n)$. Arguing similarly for other jobs implies that the total stretch is $\Omega(n^2)$, hence giving a lower bound of $\Omega(n)$ on the competitive ratio. Moreover, since all jobs arrive at $t = 0$, a speed up of k implies that the flow time reduces by k times and hence with an $O(1)$ speed up the competitive ratio is still $\Omega(n)$. The fact that neither resource augmentation nor randomization seems to help is in sharp contrast to total flow time where a $(1+\epsilon)$-speed gives a $(1+\frac{1}{\epsilon})$-competitive algorithm [46], and randomization allows us to obtain an $O(\log n)$ competitive non-clairvoyant algorithm [45, 13].

The only case where resource augmentation seems to help is when the job sizes are bounded. In this case, we show the following results:

- *SETF* is a $(1+\epsilon)$-speed, $O(\log^2 B)$-competitive algorithm for minimizing total stretch non-clairvoyantly. We also also extend the results to the problem of minimizing ℓ_p norms of stretch.

  The best lower bound with a $(1+\epsilon)$-speed processor that we know of is $\Omega(\log B)$. The above result on the competitiveness of *SETF* uses the technique of relating *SETF* to *SJF*.

- Finally, we also consider the static case, that is when all the request arrive at time $t = 0$. In this case, for the problem of minimizing the ℓ_p norms of stretch, we give algorithms with matching upper and lower bounds of $\Theta(\log B)$ on the competitive ratio.

Our results for stretch scheduling are summarized in Table 1.2 below.

Problem Scenario	Speed	*SETF*	Lower bound without speed up	Lower bound with $O(1)$ speed up
Static	1	$O(n), O(\log B)$	$\Omega(n), \Omega(\log B)$	$\Omega(n), \Omega(\log B)$
Dynamic	$(1+\epsilon)$	$O(\log^{1+1/p} B)$	$\Omega(n), \Omega(B)$	$\Omega(n), \Omega(\log B)$

Table 1.2: Summary of results for minimizing the ℓ_p norms of stretch non-clairvoyantly.

Minimizing Flow Time on Multiple Machines

Scheduling on multiple machines has been an extensive area of research for several decades and an enormous amount of literature exists of this topic. Two fundamental problems in this area whose offline complexity is very well understood by now are the following:

1. Minimizing Makespan: Given m machines and n jobs all released at time $t = 0$, assign jobs to the machines such that maximum amount of work assigned to any machine is minimized.

   Various efficient approximation schemes with running times polynomial in n and m are known for this problem [41, 38, 44].

2. Minimizing total weighted completion time: Given m machines and n jobs with arbitrary releases times, sizes and weights, find the schedule that minimizes the total weighted completion time.

   A relatively recent paper [1] gives various efficient approximation schemes for this problem (and several of its variants). Again the running time of these schemes in polynomial in n and m.

The natural extensions of these problems to the flow time case are minimizing the maximum flow time, and minimizing the total (weighted) flow time. Unfortunately, these problems are not very well understood. In this thesis, we make some progress in this direction. We begin by describing the prior work, and then our work.

We first look at minimizing the total flow time. In the online setting, a celebrated result of Leonardi and Raz [55] is that ***SRPT*** is $O(\log(\min\{\frac{n}{m}, B\}))$ competitive on identical parallel machines. They also show a matching lower bound of $\Omega(\log(\min\{\frac{n}{m}, B\}))$ on the competitive ratio on any online algorithm. The algorithm of [55] has been extended to various other restricted settings (such as eliminating migrations [6] or immediately dispatching a job to a machine as it arrives [4]). Interestingly however, even in the offline case these algorithms remain the best known algorithms for the problem. Even for the case of $m = 2$, it is not known where an $O(1)$ approximation algorithm for minimizing total flow time exists. On the other hand, the only $\mathcal{NP}$-Hardness of the problem is known. In particular, it is not even known if the problem is $\mathcal{APX}$-Hard. Obtaining an $O(1)$ approximation for this problem (even for the case of $m = 2$) has been a major open problem in scheduling [67, 55, 6].

We obtain the following results:

- We give a quasi-polynomial time approximation scheme (QPTAS) for minimizing total flow time on a constant number of machines. In particular, our algorithm produces a $(1 + \epsilon)$ approximate solution and has running time $O(n^{m \log n/\epsilon^2})$. Our result suggests that a PTAS is likely for this problem for constant m.

  While elegant approximation schemes are known for minimizing total completion time and makespan, not many results are known for flow time. One difficulty in getting efficient approximation schemes is that flow time is very sensitive to small changes in the schedule. For example, the usual techniques of rounding the job sizes or release dates to powers of $(1 + \epsilon)$ do not work. Our main idea is a technique which allows us to store the approximate state under ***SRPT*** for any subset of jobs using $O(\log^2 n)$ bits of information[4]. Moreover, this state description has sufficient information to allow us to compute the new state as time progress and jobs are worked upon or when new jobs arrive.

- Next, we consider the problem of minimizing the total weighted flow time on multiple machines. Recently, Chekuri and Khanna made the first break through for the weighted case by giving an elegant $QPTAS$ in the single machine case [21] (As mentioned earlier, there is no PTAS known for

[4]Reducing this to $O(\log n)$ would imply a polynomial time approximation scheme.

minimizing total weighted flow time even for the case of a single machine). However, their results do not carry over to $m > 1$. The main difficulty is that if we consider a particular machine, an arbitrary subset of jobs could be assigned to be processed on it. It was unclear how to encode this information in sufficient detail using few bits. Our technique above, allows us to extend the result of [21] to a constant number of machines.

- In our quasi-polynomial time approximations schemes, the running time depends exponentially on the number of machines. This seems exorbitant given the existence of approximation schemes with running times polynomial in m for the makespan and completion time problems. However, we show that this dependence on m is unlikely to be improved for the flow time problem. In particular, we show that obtaining an $n^{o(1)}$ approximation algorithm with running time $2^{polylog(n,m)}$ for the weighted flow time problem (even when all the weights are polynomially bounded in n) would imply that $\mathcal{NP} \subseteq \mathcal{DTIME}(n^{polylog(n)})$.

Finally, we consider the problem of minimizing the maximum flow time on m machines. Notice that if all the jobs are released at time 0, this reduces to the classical problem of makespan minimization, for which a lot of previous work has been done [41, 38, 54, 44]. However, minimizing the maximum flow time with general release times the only result known is a $3 - 2/m$ online algorithm [17], which in fact works for arbitrary m. Obtaining a PTAS has been open for this problem [17, 67]. We partially solve the problem by obtaining a PTAS for the special case where the number of machines m is a constant.

Combining Flow time Scheduling and Call Admission

We consider the problem of scheduling on a single machine to minimize flow time where the jobs can be rejected at some cost. Formally, for each job, the processor pays for each time unit the job is present in the system, and additionally a rejection cost is charged if this job is ever rejected. One motivation for this problem arises from trying to formalize the scheduling decisions faced by a person in real life, where one also has the flexibility to accept or reject a request. Ideally, one would like not to have too many jobs on ones list of things to do (as this causes jobs to be delayed), or equivalently, one would like to have a small flow time, and rejecting some fraction of jobs might be necessary for this to happen (but of course one cannot ignore all work one is asked to do).

Scheduling problems with rejection costs have been considered previously in other settings. In particular, there has been work on minimizing makespan with rejection [12, 68, 39] and on minimizing weighted completion time with rejections [29]. However, for the case of flow time, we do not know of any prior work.

We show the following results:

- First, we consider the simplest case where all jobs have the same rejection cost. We give a 2-competitive algorithm for this case.

- In the more general setting where the rejection costs are allowed to be arbitrary, we show that no randomized algorithm can be $n^{o(1)}$ competitive. This motivates us the consider the problem with arbitrary rejection costs in the resource augmentation setting.

- In fact, we consider a more general setting where the jobs have arbitrary weights and arbitrary rejection costs. Here a job pays its weight for each time unit it is present, and a rejection cost if this job is ever rejected. We give a $(1+\epsilon)$-speed, $O((\log W + \log C)^2)$-competitive algorithm, where C denotes the maximum to minimum rejection cost ratio.

To solve this problem, we first consider the special case where jobs have arbitrary weights but same rejection costs. We give an $O(\log^2 W)$ competitive algorithm for this case and show that the previous problem can be reduced to this while losing a factor of $(1+\epsilon)$ in the speed.

Most of the techniques used to prove the results stated above are a combination of the techniques used for the weighted flow time problem discussed previously.

Results for Completion Time Scheduling via Flow Time

Finally, we show how results for flow time problems (which possibly use excessive resource augmentation), can be used to prove results about completion time problems (without any resource augmentation). While completion time as a measure of performance is perhaps not very useful in the context of client-server systems. It is of significant academic interest and has been studied extensively.

Our contribution is a technique to transform an algorithm for a flow time problem which possibly uses resource augmentation to obtain an algorithm for the corresponding completion time problem which does not use resource augmentation. Our transformation carries online algorithms to online algorithms and also preserves non-clairvoyance. As a corollary of this result we will obtain $O(1)$-competitive online clairvoyant and non-clairvoyant algorithms for minimizing the weighted ℓ_p norms of completion time.

1.6 Related Areas

We wish to point out that the results considered in this thesis comprise a very small portion of the problems considered in scheduling theory in general. Scheduling problems encompass a variety of different optimization problems and are studied in a variety of settings such as worst case analysis and average case analysis. Some books on the subject which provide a broad overview of the results and techniques are [20, 62, 58, 25].

We now describe some topics that have been extensively studied in the computer science literature, but not considered in this thesis:

- Makespan Minimization: In these problems, typically there are multiple machines and the jobs are all available starting at time $t = 0$. The goal is to find a schedule such that all jobs can be finished by some time T. Often there are various other restrictions, such as jobs might have machine dependent processing times, there might be various dependency constraints on when a job might be executed, jobs can be migrated (i.e., same job can be executed on different machines) and so on. Another widely studied class of makespan minimization problems are what are known as shop scheduling problems. Here each job consists of various operations which must be executed sequentially. Often there are constraints on how these operations might be scheduled, these different constraints give rise to different types of “shop” scheduling problems.

- Scheduling with deadlines: This is also known as *real time* or due date scheduling. Here, in addition to release times and sizes, job also have a *deadline*, before which the job must be executed. A nice introduction to such problems can be found in [20, 26]. The goal typically in such problems is to complete the maximum number of jobs by their deadlines, or to minimize various functions of tardiness (defined as the amount by which a job exceeds its deadline).

Chapter 2

Scheduling to Minimize ℓ_p norms of Flow Time

2.1 Introduction

The algorithms Shortest Remaining Processing Time (*SRPT*) and Shortest Elapsed Time First (*SETF*) are generally regarded as the best *clairvoyant* and *nonclairvoyant* server scheduling policies for optimizing average Quality of service (QoS). *SRPT* is optimal for average flow time. *SETF* is an $(1+\epsilon)$-speed, $(1+1/\epsilon)$-competitive algorithm for average flow time [46] — no $O(1)$-competitive nonclairvoyant algorithm exists for average flow time [59] — and a randomized variation *RMLF* of *SETF* is known to be asymptotically strongly competitive amongst randomized algorithms for average flow time [13, 45, 59].

In spite of this, *SRPT* and *SETF* are generally not implemented in server systems. Apache, currently most widely used web server [43], uses a separate process for each request, and uses a First In First Out (*FIFO*) policy to determine the request that is allocated a free process slot [42]. The most commonly cited reason for adopting *FIFO* is a desire for some degree of fairness to individual jobs [35], i.e. it is undesirable for some jobs to be starved. In some sense *FIFO* is the optimally fair policy in that it optimizes the maximum flow time objective. Operating systems such as Unix don't implement pure *SETF*, or even pure *MLF*, the less preempting version of *SETF*. Once again this is out of fear of starving jobs [72, 73]. Unix systems adopt compromise policies that attempt to balance the competing demands of average QoS and fairness. In particular, Unix scheduling policies generally raise the priority of processes in the lower priority queues that are being starved [72].

The desire to optimize for the average and the desire to not have extreme outliers generally conflict. The most common way to compromise is to optimize the ℓ_p norm, generally for something like $p = 2$ or $p = 3$. For example, the standard way to fit a line to collection of points is to pick the line with minimum least squares, equivalently ℓ_2, distance to the points, and Knuth's TEXtypesetting system uses the ℓ_3 metric to determine line breaks [49, page 97]. The ℓ_p, $1 < p < \infty$, metric still considers the average in the sense that it takes into account all values, but because x^p is strictly a convex function of x, the ℓ_p norm more severely penalizes outliers than the standard ℓ_1 norm.

This leads us to consider server scheduling algorithms for optimizing the ℓ_p norms, $1 < p < \infty$, of flow time. Formally, the ℓ_p norms of flow time is defined as $\sum_i (f_i^p)^{1/p}$. We will use $F^p(A)$ to denote the sum of the p^{th} powers of flow time under an algorithm A.

In section 2.3, we show that that are no $n^{o(1)}$-competitive online server scheduling algorithms for any

Algorithm	Speed	Competitive Ratio
Any Clairvoyant Algorithm	1	$n^{\Omega(1)}$ for $1 < p < \infty$
SJF	$(1+\epsilon)$	$O(1/\epsilon)$
SRPT	$(1+\epsilon)$	$O(1/\epsilon)$
SETF	$(1+\epsilon)$	$O(1/\epsilon^{2+\frac{2}{p}})$
RR	$(1+\epsilon)$	$\Omega(n^{(1-2\epsilon p)/p})$
Any Non- clairvoyant Algorithm	$(1+\epsilon)$	-

Table 2.1: Results for ℓ_p norms of Flow Time

ℓ_p metric, $1 < p < \infty$ of flow time. Perhaps this is a bit surprising, as there are optimal online algorithms, ***SRPT*** and ***FIFO***, for the ℓ_1 and ℓ_∞ norms. This negative result motivates us to fall back to resource augmentation analysis. We first consider the standard online algorithms aimed at average QoS, that is, Shortest Job First (*SJF*), *SRPT*, and *SETF*. We show that the clairvoyant algorithms, *SJF* and *SRPT*, are $O(1+\epsilon)$-speed, $O(1/\epsilon)$-competitive for the ℓ_p norms of flow. We show that the nonclairvoyant algorithm *SETF* is $O(1+\epsilon)$-speed, $O(1/\epsilon^{(2+2/p)})$-competitive for the ℓ_p norms of flow. We summarize our results in table 2.1. Note that all of the results assume that p is constant, so that multiplicative factors, that are a function of p alone, are absorbed into the constant in the asymptotic notation.

These resource augmentation results argue that the concern, that the standard algorithms aimed at optimizing average QoS might unnecessarily starve jobs, is unfounded when the server is less than fully loaded. It might be tempting to conclude that all reasonable algorithms should have such guarantees. However, we show that this is not the case in section 2.7. More precisely, the standard Processor Sharing, or equivalently Round Robin, algorithm is not $(1+\epsilon)$-speed, $O(1)$-competitive for any ℓ_p norm of flow, $1 < p < \infty$ and for sufficiently small ϵ. This is perhaps surprising, since fairness is a commonly cited reason for adopting Processor Sharing [72]. [1]

2.2 Related Results

The results in the literature that are closest in spirit to those here are found in a series of papers, including [9, 27, 35, 65]. These papers also argue that *SRPT* will not unnecessarily starve jobs any more than Processor Sharing does under "normal" situations. In these papers, "normal" is defined as there being a Poisson distribution on release times, and processing times being independent samples from a heavily tailed distribution. More precisely, these papers argue that every job should prefer *SRPT* to Round Robin (Processor Sharing) under these circumstances. Experimental results supporting this hypothesis are also given. So informally our results and these papers reach the same conclusion about the superiority of *SRPT*. But in a formal sense the results are incomparable.

There have been some prior results on scheduling problems with ℓ_p norms. Load balancing in the ℓ_p norm is discussed in [2, 3, 5]. PTAS for ℓ_p norms of completion times, without release times, is given in [30].

A seemingly related problem is that of minimizing the weighted flow time, studied recently by [23, 21, 7]. Observe that minimizing the sum of squares of flow time is equivalent to minimizing the weighted flow

[1]The results in this chapter are from [11].

time where the weight of a job at any time is equal to its age (hence the weights are linearly increasing with time). However, the fact that the weights are changing makes our problem substantially different. For example, [7] give an $O(\log W)$ competitive algorithm for weighted flow time (which would correspond to an $O(\log n + \log B)$ competitive algorithm in our case, since the maximum weight W in our case is at most nB). However, our lower bounds in section 2.3 show that any randomized online algorithm is $\Omega(\max\{n^{1/5}, B^{1/5}\})$ competitive for minimizing flow time squared.

2.3 General Lower Bounds

Theorem 2.1 *The competitive ratio of any randomized algorithm A against an oblivious adversary for F^p, $1 < p < \infty$, is $n^{\Omega(1)}$. In particular, the competitive ratio of any randomized algorithm is $\Omega(n^{(p-1)/(p(3p-1))})$.*

Proof. We use Yao's minimax principle for online cost minimization problems [19] and lower bound the expected value of the ratio of the objective functions for A and *Opt* on input distribution which we specify. The inputs are parameterized by integers L, α, and β in the following manner. A *long* job of size L arrives at $t = 0$. From time 0 until time $L^\alpha - 1$ a job of size 1 arrives every unit of time. With probability $1/2$ this is all of the input. With probability $1/2$, $L^{\alpha+\beta}$ *short* jobs of length $1/L^\beta$ arrive every $1/L^\beta$ time units from time L^α until $2L^\alpha - 1/L^\beta$.

Let $\alpha = \frac{p+1}{p-1}$, and $\beta = 2$. We now compute $F^p(A)$ and $F^p(\textit{Opt})$. Consider first the case that A doesn't finish the long job by time L^α. Then with probability $1/2$ the input contains no short jobs. Then $F^p(A)$ is at least the flow of the long job, which is at least $L^{\alpha p}$. In this case the adversary could first process the long job and then process the unit jobs. Hence, $F^p(\textit{Opt}) = O(L^p + L^\alpha \cdot L^p) = O(L^{\alpha+p})$. The competitive ratio is then $\Omega(L^{\alpha p-\alpha-p})$, which is $\Omega(L)$ by our choice of α.

Now consider the case that A finishes the long job by time L^α. Then with probability $1/2$ the input contains short jobs. One strategy for the adversary is to finish all jobs, except for the big job, when they are released. Then $F^p(\textit{Opt}) = O(L^\alpha \cdot 1^p + L^{\alpha+\beta} \cdot (1/L^\beta)^p + L^{\alpha p})$. It is obvious that the dominant term is $L^{\alpha p}$, and hence, $F^p(\textit{Opt}) = O(L^{\alpha p})$. Now consider the subcase that A has at least $L/2$ unit jobs unfinished by time $3L^\alpha/2$. Since these unfinished unit jobs must have been delayed by at least $L^\alpha/2$, $F^p(A) = \Omega(L \cdot L^{\alpha p})$. Clearly in this subcase the competitive ratio is $\Omega(L)$. Alternatively, consider the subcase that A has at most $L/2$ unfinished jobs by time $3L^\alpha/2$. Since, the total unfinished by time $3L^\alpha/2$ is L, there must be at least $L^{1+\beta}/2$ unfinished small jobs, and hence at least $L^{1+\beta}/4$ small jobs that have been delayed for at least $L/4$ time units. By the convexity F^p, the optimal strategy for A from time $3L^\alpha/2$ onwards is to delay each small job by the same amount. Thus A delays $L^{\alpha+\beta}/2$ small jobs (that arrive during $3L^\alpha/2$ and $2L^\alpha$) by at least $L/2$. Hence in this case, $F^p(A) = \Omega(L^{\alpha+\beta} \cdot L^p)$. This gives a competitive ratio of $\Omega(L^{\alpha+\beta+p-\alpha p})$, which by the choice of β is $\Omega(L)$. ■

2.4 Analysis of SJF

In this section we show that *SJF* is a $(1+\epsilon)$-speed, $O(1/\epsilon^p)$-competitive for the F^p objective function.

We fix a time t. Let $U(\textit{SJF}, t)$ and $U(\textit{Opt}, t)$ denote the unfinished jobs at time t in *SJF* and *Opt* respectively, and $\mathcal{D} = U(\textit{SJF}, t) - U(\textit{Opt}, t)$. Let $\textit{Age}^p(X, t)$ denote the sum over all jobs $J_i \in X$ of $(t - r_i)^{p-1}$. We will demonstrate the following local competitiveness condition for all times t

$$\textit{Age}^p(\mathcal{D}, t) \leq O(1/\epsilon^p)\textit{Age}^p(U(\textit{Opt}, t), t)$$

This will establish our desired results because $F^p(A) = p\int_t \textit{Age}^p(U(A,t), t)dt$.

Before proceeding we introduce some needed notation. Let $V(t, \alpha)$ denote the aggregate unfinished work at time t among those jobs J_j that satisfy the conditions in the list α. So for example, in the next lemma we consider $V(r_i, J_j \in U(\textit{Opt}, r_i), r_j \leq r_i, p_j \leq p_i)$, the amount of work that *Opt* has left on jobs J_j that arrived before J_i, are shorter than J_i and that *Opt* has not finished by time r_i. Let $P(\alpha)$ denote the aggregate initial processing times of jobs J_j that satisfy the conditions in the list α.

We prove local competitiveness in the following manner. Let $1, \ldots, k$ denote the indices of jobs in $\mathcal{D}$ such that $p_1 \leq p_2 \ldots \leq p_k$. Consider the jobs in $\mathcal{D}$ in the order in which they are indexed. Assume that we are considering job J_i. We allocate to J_i an $\epsilon p_i/4(1+\epsilon)$ amount of work from $V(t, J_j \in U(\textit{Opt}, t), r_j \leq t - \epsilon(t-r_i)/(4(1+\epsilon)), p_j \leq p_i)$ that was previously not allocated to a lower indexed job in $\mathcal{D}$. This establishes $O(1/\epsilon^p)$ local competitiveness for F^p for the following reasons. The total unfinished work in each $J_j \in U(\textit{Opt}, t)$ is associated with $O(1/\epsilon)$ longer jobs in $\mathcal{D}$. Since the jobs J_j are $\Omega(\epsilon)$ as old as J_i, the contribution of to $\textit{Age}^p(U(\textit{Opt}, t), t)$ for J_j is $\Omega(\epsilon^{p-1})$ as large as the contribution of J_i to $\textit{Age}^p(U(\textit{SJF}, t), t)$.

We now turn to proving that this association scheme works, that is, the scheme never runs of work to assign to the jobs in $\mathcal{D}$. Consider a fixed job $J_i \in \mathcal{D}$. Let t' denote the time $t - \frac{\epsilon}{4(1+\epsilon)}(t - r_i)$. If this scheme is going to fail on job J_i then, informally, the amount of work on jobs of size $\leq p_i$ that *Opt* had left at time r_i, plus the work made up by jobs of size $\leq p_i$ that arrived during $(r_i, t']$, minus the work that *Opt* did during $(r_i, t]$, minus the work that is allocated to $J_1, \ldots, J_i$ should be negative. The amount of work in $V(t, J_j \in U(\textit{Opt}, t))$ that is allocated to $J_1, \ldots, J_i$ is at most

$$\frac{\epsilon}{4(1+\epsilon)}P(J_j \in \mathcal{D}, r_j \leq r_i, p_j \leq p_i) + \frac{\epsilon}{4(1+\epsilon)}P(J_j \in \mathcal{D}, r_j > r_i, p_j \leq p_i)$$

Moreover, as Opt can finish at most $(t - r_i)$ work during time $(r_i, t]$, it is sufficient to show that

$$\begin{aligned}
&V(r_i, J_j \in U(\textit{Opt}, r_i), r_j \leq r_i, p_j \leq p_i) + P(r_j \in (r_i, t'], p_j \leq p_i) - (t - r_i) \\
&-\frac{\epsilon}{4(1+\epsilon)}P(J_j \in \mathcal{D}, r_j \leq r_i, p_j \leq p_i) + \frac{\epsilon}{4(1+\epsilon)}P(J_j \in \mathcal{D}, r_j > r_i, p_j \leq p_i) \geq 0
\end{aligned}$$

Or equivalently,

$$\begin{aligned}
&V(r_i, J_j \in U(\textit{Opt}, r_i), r_j \leq r_i, p_j \leq p_i) + P(r_j \in (r_i, t'], p_j \leq p_i) \\
&-\frac{\epsilon}{4(1+\epsilon)}P(J_j \in \mathcal{D}, r_j \leq r_i, p_j \leq p_i) + \frac{\epsilon}{4(1+\epsilon)}P(J_j \in \mathcal{D}, r_j > r_i, p_j \leq p_i) \geq (t - r_i) \quad (2.1)
\end{aligned}$$

The rest of this section will be devoted to establishing equation 2.1. We know that

$$(t - r_i) \geq P(J_j \in \mathcal{D}, r_j > r_i, p_j \leq p_i) \tag{2.2}$$

since *Opt* had to finish all such jobs considered on the right hand side between time r_i and time t. Then by substitution, to prove equation 2.1 it suffices to prove:

$$V(r_i, J_j \in U(\textit{Opt}, r_i), r_j \leq r_i, p_j \leq p_i) + P(r_j \in (r_i, t'], p_j \leq p_i) - \frac{\epsilon}{4(1+\epsilon)} P(J_j \in \mathcal{D}, r_j \leq r_i, p_j \leq p_i) \geq (\frac{4+5\epsilon}{4(1+\epsilon)})(t - r_i) \tag{2.3}$$

We now concentrate on replacing the term $V(r_i, J_j \in U(\textit{Opt}, r_i), r_j \leq r_i, p_j \leq p_i)$ in equation 2.3. This is where the $(1+\epsilon)$ speed up is crucially used.

Lemma 2.2 *For all times u and for all values p_i,*

$$\begin{aligned} & V(u, J_j \in U(\textit{Opt}, u), r_j \leq u, p_j \leq p_i) \\ \geq\ & \frac{\epsilon}{1+\epsilon}(P(J_j \in U(\textit{SJF}, u), p_j \leq p_i) + \frac{\epsilon}{1+\epsilon} V(u, J_j \in U(\textit{SJF}, u), p_j \leq p_i) \end{aligned}$$

Proof. The proof is by induction on the time u. Whenever there is a job $J_j \in U(\textit{SJF}, u)$ with $p_j \leq p_i$, then the right hand side of the inequality decreases at least as fast as the left hand side since *SJF* has a $(1+\epsilon)$-speed processor. If the is no such job J_j, then the right hand side of the inequality is zero. ∎

Using lemma 2.2 with $u = r_i$, then to prove equation 2.3 it is sufficient by substitution to prove:

$$\begin{aligned} & \frac{\epsilon}{1+\epsilon} P(J_j \in U(\textit{SJF}, r_i), p_j \leq p_i) + \frac{1}{1+\epsilon} V(r_i, J_j \in U(\textit{SJF}, r_i), p_j \leq p_i) \\ & + P(r_j \in (r_i, t'], p_j \leq p_i) - \frac{\epsilon}{4(1+\epsilon)} P(J_j \in \mathcal{D}, r_j \leq r_i, p_j \leq p_i) \\ \geq\ & (\frac{4+5\epsilon}{4(1+\epsilon)})(t - r_i) \end{aligned} \tag{2.4}$$

Since obviously, $P(J_j \in U(\textit{SJF}, r_i), p_j \leq p_i) \geq P(J_j \in \mathcal{D}, r_j \leq r_i, p_j \leq p_i)$ and also $V(r_i, J_j \in U(\textit{SJF}, r_i), p_j \leq p_i) \leq P(J_j \in U(\textit{SJF}, r_i), p_j \leq p_i)$, to prove equation 2.4 it suffices to show that

$$\frac{4+3\epsilon}{4(1+\epsilon)} V(r_i, J_j \in U(\textit{SJF}, r_i), p_j \leq p_i) + P(r_j \in (r_i, t'], p_j \leq p_i) \geq (\frac{4+5\epsilon}{4(1+\epsilon)})(t - r_i) \tag{2.5}$$

During $[r_i, t']$, *SJF* does $(1+\epsilon)(t' - r_i)$ work on jobs shorter than J_i. By algebraic simplification $(1+\epsilon)(t' - r_i) = \frac{4+3\epsilon}{4}(t - r_i)$. The jobs that *SJF* worked on during $[r_i, t']$ either had to arrive before r_i or during $[r_i, t']$. Therefore, it trivially follows that

$$\begin{aligned} \frac{4+3\epsilon}{4}(t - r_i) &= V(r_i, J_j \in U(\textit{SJF}, r_i), p_j \leq p_i) - V(t', J_j \in U(\textit{SJF}, r_i), p_j \leq p_i) \\ & \quad + P(r_j \in (r_i, t'], p_j \leq p_i) - V(t', r_j \in (r_i, t'], p_j \leq p_i) \end{aligned}$$

By dropping the negative terms on the right hand side we get that:

$$\frac{4+3\epsilon}{4}(t - r_i) \leq V(r_i, J_j \in U(\textit{SJF}, r_i), p_j \leq p_i) + P(r_j \in (r_i, t'], p_j \leq p_i)$$

or equivalently,

$$(t - r_i) \leq \frac{4}{4+3\epsilon} V(r_i, J_j \in U(\textbf{\textit{SJF}}, r_i), p_j \leq p_i) + \frac{4}{4+3\epsilon} P(r_j \in (r_i, t'], p_j \leq p_i) \tag{2.6}$$

Hence by substitution using equation 2.6, to prove equation 2.5 it suffices to prove:

$$\begin{aligned} & \frac{4+3\epsilon}{4(1+\epsilon)} V(r_i, J_j \in U(\textbf{\textit{SJF}}, r_i), p_j \leq p_i) + P(r_j \in (r_i, t'], p_j \leq p_i) \\ \geq \; & \frac{4+5\epsilon}{(4+3\epsilon)(1+\epsilon)} \left[V(r_i, J_j \in U(\textbf{\textit{SJF}}, r_i), p_j \leq p_i) + P(r_j \in (r_i, t'], p_j \leq p_i) \right] \end{aligned} \tag{2.7}$$

Now, it is easy to see that equation 2.7 is true since,

$$1 \geq \frac{4+5\epsilon}{(4+3\epsilon)(1+\epsilon)}$$

and

$$\frac{4+3\epsilon}{4(1+\epsilon)} \geq \frac{4+5\epsilon}{(4+3\epsilon)(1+\epsilon)}$$

Thus we have proved our main theorem for this section.

Theorem 2.3 *SJF is $(1+\epsilon)$-speed, $O(\frac{1}{\epsilon})$-competitive for the ℓ_p norms of flowtime, for $p \geq 1$.*

2.5 Analysis of *SRPT*

In this section we prove identical results for ***SRPT***. The analysis is somewhat more involved than that for ***SJF*** because we need to handle the remaining times of jobs under ***SRPT*** more carefully.

Let us define the *relaxed age*, denoted by $rAge^p(J_i, t)$, of a job J_i at time t to be 0 if $(t - r_i) \leq 8p_i/\epsilon$ and $(t - r_i)^{p-1}$ otherwise.

Note that if $F(J_i) \leq 8p_i/\epsilon$, then $p \int_t rAge^p(J_i, t) = 0$ and if $F(J_i) > 8p_i/\epsilon$, then $p \int_t rAge^p(J_i, t) = F^p(J_i) - (8p_i/\epsilon)^p$. Thus, we always have that

$$F^p(J_i) \leq p \int_t rAge^p(J_i, t) + (8p_i/\epsilon)^p$$

Define $rAge(X, t)$ and $rSAge(X, t)$ for a set of jobs X as the sum of $rAge$'s and $sAge$'s of jobs in X. Now, if we show that for all t,

$$rAge^p(U(SRPT, t) \setminus U(Opt, t), t) \leq O(1/\epsilon^p) Age^p(U(Opt, t), t)$$

then it follows that,

$$rAge^p(U(SRPT, t), t) = O(1/\epsilon^p) Age^p(U(Opt, t), t)$$

and hence that

$$\int_t rAge^p(U(SRPT, t), t) = O(1/\epsilon^p) \int_t Age^p(U(Opt, t), t)$$

Now since the flow time of J_i it at least p_i, it is easy to see that $F^p(SRPT) \le \int_t rAge^p(U(SRPT,t),t) + \sum_i (8p_i/\epsilon)^p = O(1/\epsilon)^p F^p(Opt)$.

To prove these, we modify the analysis for *SJF* as follows: Given a time t, define $\mathcal{D}$ to consist of only those jobs which have age more that $8/\epsilon$ times their size, and which Opt has finished but are present under *SRPT*. Mimicking the proof for *SJF*, for every job J_i in $\mathcal{D}$ will we show that we can associate $\epsilon p_i/4(1+\epsilon)$ units of work (which has not been already allocated to a lower indexed job in $\mathcal{D}$) from some job of size $\le p_i$ in Opt and which has age at least $\frac{\epsilon(t-r_i)}{4(1+\epsilon)}$. Clearly, this implies that

$$rAge^p(U(SRPT,t) - U(Opt,t),t) \le O(1/\epsilon^p)Age^p(U(Opt,t),t)$$

For job J_j, let $rem(j,t)$ denote its remaining processing requirement at time t. We begin by a result similar to Lemma 2.2.

Lemma 2.4 *For all times u and values of p_i,*

$$\begin{aligned} V(u, J_i \in U(\textbf{Opt}, u), r_j \le u, p_j \le p_i) &\ge \frac{\epsilon}{1+\epsilon} P(J_j \in U(\textbf{SRPT}, u), p_j \le p_i) \\ &+ \frac{1}{1+\epsilon}[V(u, J_j \in U(\textbf{SRPT}, u), p_j \le p_i) - p_i] \end{aligned} \tag{2.8}$$

Proof. First observe that by the nature of SRPT, at any time u and any value p, there can be at most one which has size greater than p and remaining time less than p. Thus the term

$$V(u, J_j \in U(\textbf{SRPT}, u), rem(j,u) \le p_i, p_j > p_i)$$

never exceeds p_i.

We now prove the lemma by induction on the time u. Whenever there is a job in with size less than p_i or remaining time less than p_i then the right hand side of the inequality decreases at least as fast as the left hand side. If the is no such job J_j, then the right hand side of the inequality is $-p_i$ where as the left hand side is 0. Now, it might happen that SRPT works on some job of size $> p_i$ and its remaining time becomes p_i or less. However, it is easy to see that in this case the right hand side is no more than 0. ■

We now apply Equation 2.8 with $u = r_i$ to Lemma 2.3. We also use the obvious facts that

$$P(J_j \in U(\textbf{SRPT}, r_i), p_j \le p_i) \ge P(J_j \in \mathcal{D}, r_j \le r_i, p_j \le p_i)$$

and

$$P(J_j \in U(\textbf{SRPT}, r_i), p_j \le p_i) \ge V(r_i, J_j \in U(\textbf{SRPT}, r_i), p_j \le p_i)$$

Thus it suffices to show that

$$\begin{aligned} &\frac{4+3\epsilon}{4(1+\epsilon)} P(J_j \in U(\textbf{SRPT}, r_i), p_j \le p_i) + P(r_j \in (r_i, t'], p_j \le p_i) - \frac{1}{1+\epsilon} p_i \\ \ge\ &(\frac{4+5\epsilon}{4(1+\epsilon)})(t - r_i) \end{aligned} \tag{2.9}$$

Since SRPT does not finish J_i by time t (and hence by time t'). It must be the case that $(1+\epsilon)(t'-r_i) \leq V(r_i, J_j \in U(\textbf{\textit{SRPT}}, r_i), rem(j) \leq p_i) + P(r_j \in (r_i, t'], p_j \leq p_i)$. Since,

$$V(r_i, J_j \in U(\textbf{\textit{SRPT}}, r_i), rem(j) \leq p_i) \leq V(r_i, J_j \in U(\textbf{\textit{SRPT}}, r_i), p_j \leq p_i) + p_i$$

we get that

$$\frac{4+3\epsilon}{4}(t-r_i) \leq V(r_i, J_j \in U(\textbf{\textit{SRPT}}, r_i), p_j \leq p_i) + P(r_j \in (r_i, t'], p_j \leq p_i) + p_i$$

Now, since $p_i \leq \frac{\epsilon}{8}(t-r_i)$ (this is where we use that we are considering relaxed ages) we have that,

$$\frac{4+2\epsilon}{4}(t-r_i) \leq V(r_i, J_j \in U(\textbf{\textit{SRPT}}, r_i), p_j \leq p_i) + P(r_j \in (r_i, t'], p_j \leq p_i) - p_i \tag{2.10}$$

Using 2.10, to prove 2.9 it suffices to show that

$$\begin{aligned} & \frac{4+3\epsilon}{4(1+\epsilon)} V(J_j \in U(\textbf{\textit{SRPT}}, r_i), p_j \leq p_i) + P(r_j \in (r_i, t'], p_j \leq p_i) - \frac{1}{1+\epsilon} p_i \\ \geq & \left(\frac{4+5\epsilon}{2(1+\epsilon)(2+\epsilon)}\right) \left[V(r_i, J_j \in U(\textbf{\textit{SRPT}}, r_i), p_j \leq p_i) + P(r_j \in (r_i, t'], p_j \leq p_i)\right] - p_i \end{aligned} \tag{2.11}$$

However, comparing term by term, it is easy to see that equation 2.11 always holds. Thus we get that,

Theorem 2.5 *SRPT is $(1+\epsilon)$-speed, $O(\frac{1}{\epsilon})$-competitive for the ℓ_p norms of flowtime, for $p \geq 1$.*

2.6 Flow Analysis of SETF

Our goal is to show that *SETF* is $(1+\epsilon)$-speed, $O(1/\epsilon^{2+2/p})$-competitive for the ℓ_p norm of flowtime. Recall that *SETF* is the non-clairvoyant algorithm that at any time works on the job that has received the least amount of processing thus far. It can be seen that the local-competitiveness technique used in the previous sections on clairvoyant scheduling does not work in the non-clairvoyant case. To prove our result, we will consider a series of intermediate steps where we relate various scheduling algorithms to each other, while losing a $(1+\epsilon)$ factor speed up at each step.

There are two steps in the analysis: Suppose first that we knew the value of ϵ (i.e. our non-clairvoyant algorithm could use the value of ϵ while scheduling), for this case we give a non-clairvoyant algorithm that we call *MLF* (described below) and relate it to *Opt*. We show that if *MLF* has a $(1+\epsilon)$ times faster processor then $F^p(\textit{MLF}) = O(1/\epsilon^{2p+2})F^p(\textit{Opt})$. Next, we show that we can remove our dependence on the knowledge of ϵ. To do this we show that the performance of *SETF* with a $(1+\epsilon)$ faster processor is no worse than that of *MLF*. Since *SETF* does not require any knowledge of ϵ, this will imply the result.

To compare *MLF* with *Opt* we will introduce a somewhat general technique which will also be used later in Chapter 4 to prove results about minimizing the ℓ_p norms of stretch non-clairvoyantly. The main idea is the following: We first relate the behavior of *MLF* on an arbitrary instance $\mathcal{J}$ to the behavior of *SJF* on another instance $\mathcal{L}$ derived from $\mathcal{J}$. The crucial step is then to relate the behavior of *SJF* on $\mathcal{L}$ to the behavior of *SJF* on $\mathcal{J}$. This reduces the problem of relating *MLF* to *Opt* to that of relating *SJF* to *Opt*. The latter problem is usually easier. For our problem, we will simply use the result from the previous section that *SJF* is $(1+\epsilon)$-speed, $O(1/\epsilon)$-competitive for minimizing ℓ_p norms of flow time. We now describe the details.

Our *MLF* is a variation of the standard Multilevel Feedback Queue algorithm where the quanta for the queues are set as a function of ϵ. Let $\ell_i = \epsilon((1+\epsilon)^i - 1)$, $i \geq 0$, and let $q_i = \ell_{i+1} - \ell_i = \epsilon^2(1+\epsilon)^i$, $i \geq i$. In *MLF* a job J_j is in *level k at time t* if the work done on J_j by time t is $\geq \ell_k$ and $< \ell_{k+1}$. *MLF* maintains the invariant that it is always running the earliest arriving job in the smallest nonempty level.

Let *SETF*$_s$ (resp. *MLF*$_s$) denote the algorithms *SETF* and *MLF* executed with an s speed processor. We first show the relation between *SETF* and *MLF*.

Lemma 2.6 *For any instance $\mathcal{I}$, and for any $J_j \in \mathcal{I}$, J_j completes in $SETF_{1+\epsilon}$ before J_j completes in MLF$_1$.*

Proof. For any job $J_j \in \mathcal{I}$ and any time t, let $w(j,t)$ denote the work done on J_j by time t. For a job with $w(j,t) \leq x$, let $\mathit{rem}_{\leq x}(j,t) = \min(p_j, x) - w(j,t)$, that is, the amount of work that must be done on J_j until either J_j completes or until J_j has been processed for x units. Let $U(A, t, w(j,t) \leq x)$ denote the set of unfinished jobs in algorithm A at time t which have less than x work done on them. Let $W_{\leq x}(A,t)$ denote $\sum_{J_j \in U(A,t,w(j,t)\leq x)} \mathit{rem}_{\leq x}(j,t)$. Now, as *SETF* is always working on the job with least work done, it is easy to see that for all x and for any algorithm A $W_{\leq p}(\mathit{SETF}, t) \leq W_{\leq p}(A,t)$.

Suppose to reach a contradiction that there is some job J_j which completes earlier in *MLF*$_1$ than in *SETF*$_{1+\epsilon}$. Clearly *MLF*$_1$ must finish the work $W_{\leq p_j}(\mathit{MLF}_1, r_j)$ before time $C_j^{\mathit{MLF}_1}$ since *MLF*$_1$ will either complete or give p_j units of processing by time $C_j^{\mathit{MLF}_1}$ to jobs arriving before r_j. Moreover, at time $C_j^{\mathit{MLF}_1}$ all the jobs in $U(\mathit{MLF}_1, t)$ have at least $p_j/(1+\epsilon)$ amount of work done on them, otherwise J_j wouldn't be run by *MLF*$_1$ at time $C_j^{\mathit{MLF}_1}$ since there would be a lower level job than J_j at this time. Consider the schedule $A_{1+\epsilon}$ that at all times t, runs the job that *MLF*$_1$ is running at time t (and idles if this job is already completed). Hence, $A_{1+\epsilon}$ would have completed J_j, and all previously arriving jobs with processing time less that p_j, by time $C_j^{\mathit{MLF}_1}$ since $A_{1+\epsilon}$ has a $(1+\epsilon)$-speed processor. Hence, by the property of *SETF* from the previous paragraph, *SETF*$_{1+\epsilon}$ completes J_j by time $C_j^{\mathit{MLF}_1}$, which is our contradiction. ■

Lemma 2.6 implies that

$$F^p(\mathit{SETF}(\mathcal{I}), s) \leq F^p(\mathit{MLF}(\mathcal{I}), s/(1+\epsilon)) \tag{2.12}$$

Now our goal will be to relate *MLF* and *Opt*. Let the original instance be $\mathcal{I}$. Let $\mathcal{J}$ be the instance obtained from $\mathcal{I}$ as follows. Consider a job J_j and let i be the smallest integer such that $p_j + \epsilon \leq \epsilon(1+\epsilon)^i$. The processing time of J_j in $\mathcal{J}$ is then $\epsilon(1+\epsilon)^i$. Let $\mathcal{K}$ be the instance obtained from $\mathcal{J}$ by decreasing each job size by ϵ. Thus, each job in $\mathcal{K}$ has size $\ell_k = \epsilon((1+\epsilon)^k - 1)$ for some k. Note that in this transformation from $\mathcal{I}$ to $\mathcal{K}$, the size of a job doesn't decrease, and it increases by at most a factor of $(1+\epsilon)^2$. Since *MLF* is has the property that increasing the length of a particular job will not decrease the completion time of any job, we can conclude that

$$F^p(\textit{MLF}(\mathcal{I}), s/(1+\epsilon)) \leq F^p(\textit{MLF}(\mathcal{K}), s/(1+\epsilon)) \tag{2.13}$$

Finally we create an instance $\mathcal{L}$ by replacing each job of size $\epsilon((1+\epsilon)^k - 1)$ job in $\mathcal{K}$ by k jobs of size $q_0, q_1, \ldots, q_{k-1}$. Note that $\ell_k = q_0 + q_1 + \ldots, q_{k-1}$. For a job of $J_j \in \mathcal{K}$, we denote the corresponding jobs in $\mathcal{L}$ by $J_{j_0}, J_{j_1}, \ldots, J_{j_{k-1}}$.

Notice that any time t, *SJF*($\mathcal{L}$) is working on a job $J_{j_b} \in \mathcal{L}$ if and only if *MLF*($\mathcal{K}$) is working on job $J_j \in \mathcal{K}$ that is in level b at time t. In particular, this implies that the completion time of J_j in *MLF*($\mathcal{K}$) is exactly the completion time of some job $J_{j_b} \in$ *SJF*($\mathcal{L}$). Hence,

$$F^p(\textit{MLF}(\mathcal{K}), s/(1+\epsilon)) \leq F^p(\textit{SJF}(\mathcal{L}), s/(1+\epsilon)) \tag{2.14}$$

By Theorem 2.3, we know that

$$F^p(\textit{SJF}(\mathcal{L}), s/(1+\epsilon) = O(1/\epsilon^p) F^p(\textit{Opt}(\mathcal{L}), s/(1+\epsilon)^2) \tag{2.15}$$

We relate the optimal schedule for $\mathcal{L}$ back to the optimal schedule for $\mathcal{I}$. To do this we first relate $\mathcal{L}$ to $\mathcal{J}$ as follows. Let $\mathcal{L}(k)$ denote the instance obtained from $\mathcal{J}$ by multiplying each job size in $\mathcal{J}$ by $\epsilon/(1+\epsilon)^k$. Next, we remove from $\mathcal{L}(k)$ any job whose size is less than ϵ^2. We claim that $\mathcal{L} = \mathcal{L}(1) \cup \mathcal{L}(2) \cup \ldots$. To see this, let us consider some job $J_j \in \mathcal{J}$ of size $\epsilon(1+\epsilon)^i$. Then, $L(1)$ 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 $\mathcal{L}(1) \cup \mathcal{L}(2) \cup \ldots$. Summarizing, we have that the $\mathcal{L}(k)'s$ are geometrically scaled down copies of J and that $\mathcal{L}$ is exactly the union of these $\mathcal{L}(k)'s$.

Our idea at a high level is as follows: Given a *good* schedule of $\mathcal{J}$, we obtain a good schedule for $\mathcal{L}(k)$. This will be easy to do as $\mathcal{L}(k)$ is a scaled down version of $\mathcal{J}$. Finally, we will superimpose the schedules for each $\mathcal{L}(k)$ to obtain a schedule for $\mathcal{L}$. This will give us a procedure to obtain a *good* schedule for $\mathcal{L}$ given a good schedule for $\mathcal{J}$. To put everything in place, we need the following lemma which relates $\mathcal{J}$ and $\mathcal{L}(k)$.

Lemma 2.7 *Let $\mathcal{L}(k)$ be as defined above. Let $F(s, J_i, \mathcal{G}))$ denote the flow time of job $J_i \in \mathcal{G}$ when SJF is run on $\mathcal{G}$ with a speed s processor. Then, for all $x \geq 1$,*

$$F(\epsilon(1+\epsilon)^{-k} \cdot x \cdot s, J_{i_k}, \mathcal{L}(k)) \leq \frac{1}{x} F(s, J_i, \mathcal{J})$$

Proof. We first show that for all jobs $J_j \in \mathcal{J}$, $F(s, J_j, \mathcal{J}) \leq (1/s) F(1, J_j, \mathcal{J})$. Let $w(x, s, J_j)$ denote the work done on job J_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_j and all times t, $w((t-r_j)/s, s, J_j) \geq w(t - r_j, 1, J_j)$.

Consider some instance where this condition is violated. Let J_j be the job and t be the earliest time for which $w((t-r_j)/s, s, J_j) < w(t - r_j, 1, J_j)$. Clearly, *SJF*$_s$ is not working on J_j at time t, due to minimality of t. Thus, *SJF*$_s$ is working on some other smaller job i. Since *SJF*$_1$ is not working on i, *SJF*$_1$ has already finished i by some time $t' < t$. However, this means that $w((t' - r_i), s, J_i) < w(t' - r_i, 1, J_i)$, which contradicts the minimality of t.

We now show the main result of the lemma. It is easy to see that the flow time of every job $J_j(k) \in \mathcal{L}(k)$ (where $J_j(k)$ is job corresponding to $J_j \in \mathcal{J}$ but scaled down by $\epsilon(1+\epsilon)^{-k}$ times) under *SJF* with a speed $\epsilon(1+\epsilon)^{-k}$ processor is at most that of the corresponding job $J_j \in \mathcal{J}$, under SJF with a unit speed processor. Thus by the fact above, running *SJF* on $\mathcal{L}(k)$ with an $x \cdot \epsilon(1+\epsilon)^{-k}$-speed processor yields a $1/x$ times smaller flow time for each job in $\mathcal{L}(k)$ than the corresponding job in $\mathcal{J}$.

■

Lemma 2.8

$$F^p(\textit{Opt}(\mathcal{L}), 1+2\epsilon) = O(1/\epsilon^2) F^p(\textit{SJF}(\mathcal{J}), 1)$$

Proof. Given the schedule *SJF*($\mathcal{J}$), we construct a following schedule A for $\mathcal{L}$ as follows. The jobs in $\mathcal{L}(k)$ are run with a speed $x_k = \epsilon(1+\frac{\epsilon}{1+\epsilon})^{-k}$ processor using the algorithm *SJF*. Note that the total speed required by A is at most

$$\sum_{i=1}^{\infty} x_i = \sum_{i=1}^{\infty} \epsilon \frac{1}{(1+\frac{\epsilon}{1+\epsilon})^i} = \frac{\epsilon}{1-\frac{1+\epsilon}{1+2\epsilon}} = 1+2\epsilon$$

By Lemma 2.7, $F^p(A(\mathcal{L}(k)), 1+2\epsilon)$ will be at most $\left(\frac{\epsilon(1+\epsilon)^{-k}}{x_k}\right)^p = \left(\frac{(1+2\epsilon)}{(1+\epsilon)^2}\right)^{kp}$ times $F^p(\textit{SJF}(\mathcal{J}, 1)$. Hence,

$$\begin{aligned}
\frac{F^p(A(\mathcal{L}), 1+2\epsilon)}{F^p(\textit{SJF}(\mathcal{J}), 1)} &\leq \sum_{i=1}^{\infty} \left(\frac{1+2\epsilon}{(1+\epsilon)^2}\right)^{ip} \\
&\leq \sum_{i=0}^{\infty} \left(1 - \frac{\epsilon^2}{(1+\epsilon)^2}\right)^{i} \\
&= O(1/\epsilon^2)
\end{aligned}$$

The proof then follows because *Opt* is at least as good as A. ■

Hence by Lemma 2.8, Theorem 2.3, and the fact that that jobs lengths in $\mathcal{J}$ are at most $(1+\epsilon)^2$ times as long as they are in $\mathcal{I}$, we get that

$$\begin{aligned}
& F^p(\textit{Opt}(\mathcal{L}), s/(1+\epsilon)^2) \\
= \ & O(1/\epsilon^2) \cdot F^p(\textit{SJF}(\mathcal{J}), \frac{s}{(1+2\epsilon)(1+\epsilon)^2}) \\
= \ & O(1/\epsilon^{p+2}) \cdot F^p(\textit{Opt}(\mathcal{J}), \frac{s}{(1+2\epsilon)(1+\epsilon)^3}) \\
= \ & O(1/\epsilon^{p+2}) \cdot F^p(\textit{Opt}(\mathcal{I}), \frac{s}{(1+2\epsilon)(1+\epsilon)^5})
\end{aligned} \tag{2.16}$$

By stringing together the inequalities 2.12, 2.13, 2.14, 2.15, 2.16, we find that

$$F^p(\textit{SETF}(\mathcal{I}), s) = O(1/\epsilon^{2p+2}) F^p(\textit{Opt}(\mathcal{I}), \frac{s}{(1+2\epsilon)(1+\epsilon)^4})$$

Hence, we obtain the result that

Theorem 2.9 *SETF is $(1+\epsilon)$-speed $O(1/\epsilon^{2+2/p})$-competitive non-clairvoyant online algorithm for minimizing the ℓ_p norm of flowtime.*

2.7 Lower Bound for Round Robin

We now show that for every $p \geq 1$, there is an $\epsilon > 0$ such that Round Robin (*RR*) is not an $(1+\epsilon)$-speed, $n^{o(1)}$-competitive algorithm for the ℓ_p norm of flow time. In particular we show that

Theorem 2.10 *For any $p \geq 1$, there is an $\epsilon > 0$ such that even with a $(1+\epsilon)$ times faster processor RR is not $n^{o(1)}$ competitive for the ℓ_p norms of flow time. In particular, for $\epsilon < 1/2p$, RR is $(1+\epsilon)$-speed, $\Omega(n^{(1-2\epsilon p)/p})$-competitive for the ℓ_p norms of flow time.*

Proof. Suppose $0 < \epsilon < 1/2p$ and *RR* has a processor of speed $(1+\epsilon)$. Consider the following instance. Two jobs of size $p_0 = 1$ arrive at $r_0 = 0$. Next we have a collection of n jobs whose release times and sizes are defined as follows. The first job of size $p_1 = p_0(1 - \frac{1+\epsilon}{2})$ arrives at time $r_1 = p_0$. In general the i^{th} job has size $p_i = (1 - \frac{1+\epsilon}{i+1})p_{i-1}$ and $r_i = \sum_{j=0}^{i-1} p_j$. The instance has the following properties which are easily verified:

1. Except for one job of size 1 which arrives at time 0, each job under SRPT has a flow time equal to its size. The job of size 1 has flow time $t' = 1 + \sum_{j=0}^{n} p_j$.
2. Under *RR* with a $(1+\epsilon)$-speed processor, at each time r_i, all jobs (including the newly introduced job) have exactly the same remaining processing time. In particular this means that, all the jobs keep accumulating and finish simultaneously at time $t = (2p_0 + \sum_{j=1}^{n} p_i)/(1+\epsilon)$.

We now consider the relevant quantities. First observe that $p_i = \Pi_{j=1}^{i} \frac{(j-\epsilon)}{j+1} = \frac{1}{i+1}\Pi_{j=1}^{i}(1-\epsilon/j)$. Using the fact that for all $x \geq 0$, $1 - x \leq e^{-x}$ we get that $p_i \leq \frac{1}{i+1}e^{-H(i)\epsilon}$ where $H(i)$ is the i^{th} Harmonic number. Finally using that $H(i) \geq \ln i$, we get that $p_i \leq \frac{1}{i+1}i^{-\epsilon} \leq i^{-(1+\epsilon)}$.

We now upper bound $F^p(\textit{SRPT}, 1)$, and hence $F^p(\textit{Opt}, 1)$. Observe that the flow time of the the jobs that finishes the last is $1 + \sum_{j=0}^{n} p_j \leq \sum_{j=0}^{\infty} p_j \leq \sum_{j=0}^{\infty} j^{-(1+\epsilon)} = O(1)$. For all other jobs, their flow time is exactly equal to their processing time, and hence the sum of the p^{th} powers of the the flow time of all these jobs is $\sum_{j=0}^{n} p_j^p \leq \sum_{j=0}^{\infty} j^{-(1+\epsilon)p} = O(1)$, thus we get that $F^p(\textit{SRPT}, 1) = O(1)$.

On the other hand, in *RR* each job with size p_i has flow time at least $(i+2)p_i$ (since this job time-shares with at least $i+2$ jobs throughout its execution). We now lower bound p_i. Using the fact that $e^{-2x} \leq 1 - x$ for $x \leq \frac{1}{2}$, we get that $p_i = \frac{1}{i+1}\Pi_{j=1}^{i}(1 - \epsilon/j) \geq \frac{1}{i+1}e^{-2\epsilon H(i)}$. Since $H(i) \leq \ln i + 1$ we get, $p_i \geq \frac{1}{i+1}(ei)^{-2\epsilon} = \Omega(i^{-(1+2\epsilon)})$. Thus $(i+2)p_i = \Omega(i^{-2\epsilon})$. This gives us that $F^p(\textit{RR}, 1+\epsilon)$ is at least $\sum_{i=1}^{n} i^{-2\epsilon p}$ which is $\Omega(n^\delta)$ for $2\epsilon p \leq 1 - \delta$. ∎

As a simple corollary, Theorem 2.10 implies that for $p = 2$, then RR is not O(1)-competitive even with an $(1+\epsilon)$ speed up if $\epsilon < \frac{1}{4}$.

2.8 Concluding Remarks

It is interesting to note that the schedule produced by SJF using an $(1+\epsilon)$-speed processor is $O(1)$ competitive for all ℓ_p norms simultaneously. On the other hand if no resource augmentation is allowed, one can show (by slightly modifying the example in Theorem 2.1) that there does not exist single schedule for which the ℓ_p norm of flow time is within $n^{o(1)}$ times that of the optimum schedule for all values of p. Hence, if one considers the offline problem of trying to find an $n^{o(1)}$ approximate schedule, then there has to be a different schedule for each value of p. In this sense, this is another nice property of resource augmentation that it allows us to prove a statement about a single schedule, which is otherwise not possible.

The results of this chapter for the clairvoyant case have recently been extended to the case of multiple machines by Chekuri, Khanna and Kumar [22]. They show that a simple load balancing scheme developed by Avrahami and Azar [4] is a $(1+\epsilon)$-speed, $O(1/\epsilon)$-competitive algorithms for minimizing the ℓ_p norms of flow time and stretch on multiple machines.

Some problems that remain open are the following:

Open Problem 2.1 What is the offline complexity of minimizing the ℓ_p norms of flowtime? Is the problem $\mathcal{NP}$-Hard?

Open Problem 2.2 Given a fixed p, $1 < p < \infty$. Is there a polynomial time algorithm with approximation ratio $n^{o(1)}$ for minimizing the ℓ_p norm of flow time?

Chapter 3

Scheduling to Minimize Weighted Flow Time

3.1 Introduction

Often in server systems, not all jobs are treated equally. Some jobs might be more important than others. For example, the Unix operating system has a priority parameter for jobs which can be set by the system call *nice* [72]. Moreover, some operations such as keystrokes, or other input-output operations inherently receive a higher priority as compared with other background or passive operations. Similarly in the proposed differentiated services architecture for the Internet (known as Diffserv) [50], clients are classified into premium or economy class based on the type of service level desired by them. The goal in such a setting is to give better performance to the clients in higher priority classes, possibly at the expense of those in lower priority classes.

Perhaps the simplest and most natural way to formalize setting with different priority classes is to assign weights to jobs and then consider some weighted function of the flow time or a related measure that one is interested in. In this chapter, we consider algorithms for several variants of the weighted flow time measure.

We first consider the most basic problem of minimizing the total weighted flow time on a single machine. While ***SRPT*** is optimal for minimizing the total unweighted flow time, the weighted case is known to be $\mathcal{NP}$-Hard [53]. The first non-trivial result for this problem was due to Chekuri, Khanna and Zhu [23] who gave an $O(\log^2 B)$ competitive semi-online[1] algorithm, where B is the ratio of the maximum and minimum processing times of the jobs. They also show a lower bound of 1.61 on the competitive ratio of any online algorithm. However, the work of [23] left a questions open. First, is there an $O(1)$ competitive algorithm for minimizing weighted flow time with arbitrary weights? A simpler problem left open was whether there is an $O(1)$ competitive algorithm even if there are only two different job weights? Finally, they also ask if there a "truly" online algorithm with a non-trivial competitive ratio.

Our main result is the first "truly" online algorithm that is k competitive if the job weights belong to at most k different arbitrary weight classes. Our result also gives the first $O(1)$ competitive algorithm for the case when there are $O(1)$ different job weights. As a corollary, this gives a $O(\log W)$ competitive algorithm, since we can simply round the weights up to a power of 2 which affects the quality of the solution by a factor of at most 2. While the bounds of our algorithm are in terms of W, we also show how this algorithm can be modified to give a semi-online algorithm with competitive ratio $O(\log n + \log B)$. This modified algorithm

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

is semi-online in the sense that it assumes the knowledge of n,B and W while scheduling the jobs.

Next we consider the more general problem of minimizing the weighted ℓ_p norms of flow time (i.e. $(\sum_i w_i f_i^p)^{1/p}$). Our motivation arises in trying to formalize the goals in CPU scheduling that arise in a typical operating system such as Unix:

Goal A: Amongst jobs of the same priority, there should be some balance between optimizing for average QoS and optimizing for worst case QoS.

Goal B: Higher priority jobs should get a greater share of the CPU resources, but lower priority jobs should not be starved.

We consider this problem both in the clairvoyant and non-clairvoyant setting. The impossibility of achieving an $n^{o(1)}$ competitive ratio for minimizing the weighted ℓ_p norms of flow time in the clairvoyant setting (and hence the non-clairvoyant setting), for $1 < p < \infty$, follows from Theorem 2.1. Thus we consider these problems in the resource augmentation model. We show that HDF, the natural generalization of SJF to the weighted case, is $(1+\epsilon)$-speed, $O(1/\epsilon^2)$-competitive for minimizing the weighted ℓ_p norms of flow time in the clairvoyant setting. Similarly, $WSETF$, the natural generalization of $SETF$ is $(1+\epsilon)$-speed, $O(1/\epsilon^{2+2/p})$-competitive for minimizing the weighted ℓ_p norms of flow time in the non-clairvoyant setting. For the special case of minimizing total weighted flow time non-clairvoyantly (i.e., $p = 1$), the general results above give only a $(1+\epsilon)$-speed, $O(1/\epsilon^4)$-competitive algorithm. However, we can improve the result to give a $(1+\epsilon)-$speed, $(1+1/\epsilon)$-competitive algorithm[2] The results for the weighted ℓ_p norms of flow time are summarized in Table 3.1 below:

Algorithm	Speed	p	Competitive Ratio
HDF	$1+\epsilon$	$p=1$	$1+1/\epsilon$ [16]
HDF	$1+\epsilon$	$1<p<\infty$	$O(1/\epsilon^2)$
WSETF	$1+\epsilon$	$p=1$	$1+1/\epsilon$
WSETF	$1+\epsilon$	$1<p<\infty$	$O(1/\epsilon^{2+2/p})$

Table 3.1: Resource augmentation results for the weighted ℓ_p norms of flow time.

The results for the case $1 < p < \infty$ in Table 3.1 above are obtained from results in Chapter 2. To do this, we introduce a general technique where the weighted problem is reduced to the unweighted case at the expense of a $(1+\epsilon)$ faster processor and a $1/\epsilon$ factor additional loss in the competitive ratio.

This chapter is organized as follows: We first describe the related work in Section 3.2. In Section 3.3 we give some intuition for the weighted flow time problem. In particular, we describe some simple algorithms and show why they do not perform well. In Section 3.4 we present and analyze our $O(\log W)$ competitive algorithm for weighted flow time. We then consider algorithms for the weighted ℓ_p norms of flow time in Section 3.5 and finally conclude with some open problems in Section 3.6. [3]

[2] This result first appeared in [7], however we give a much simpler proof of this result in this thesis.

[3] The results in this chapter are from [7] and [10].

3.2 Related Work

The first non-trivial result for the problem of minimizing weighted flow time on a single machine was due to Chekuri, Khanna and Zhu [23] who gave an $O(\log^2 B)$ competitive semi-online (it requires the knowledge of B beforehand) algorithm. They also show a lower bound of 1.618 on the competitiveness of any online algorithm. If resource augmentation is allowed, it is known that *HDF* is a $(1+\epsilon)$-speed, $O(1/\epsilon)$-competitive algorithm [16].

In the context of offline approximation, Chekuri and Khanna [21] give a $(1+\epsilon)$ approximation algorithm which has running time $O(n^{O(\ln W \ln B/\epsilon^3)})$. Thus, their result gives a quasi-polynomial time approximation scheme (QPTAS) when both W and B are polynomially bounded in n. Moreover, they also give a QPTAS for the case when only one of either W or B is polynomially bounded in n. Interesting however, no polynomial time algorithm with $o(n)$ approximation ratio is known for the case of arbitrary W and B.

No prior work is known for the problem of minimizing the weighted ℓ_p norms of flow time.

3.3 Preliminaries

To get a feel for minimizing weighted flow time, we first describe some simple algorithms and give reasons for their poor performance. Next, we show by an example that the algorithm of Chekuri *et al.* [23] has a competitive ratio of $\Omega(\log B)$ even in the case when the job weights are either 1 or 2. This motivates the goal of an $O(1)$ competitive algorithm for a constant number of weight classes. We begin by showing that local-competitiveness is necessary for obtaining a good algorithm for weighted flow time.

Local Competitiveness is Necessary:

Let O be the optimum algorithm and A some other algorithm. Let $W_o(t)$ and $W_a(t)$ denote the total weight of jobs present in O's and A's system respectively at time t. It is easy to see that for any algorithm A, the total weighted flow time under A can also be expressed as $\int_0^\infty W_a(t)dt$. $W_a(t) \le cW_o(t)$ for some c at all times t (i.e. locally competitive), then clearly A is c-competitive. It turns out that local-competitiveness is also necessary for obtaining a globally competitive algorithm [16]. The idea is that if $W_a(t) > cW_o(t)$ at some time t, then the adversary can start giving a stream of jobs of size $\epsilon \to 0$ and weight 1 every ϵ units of time. This forces both O and A to work on the new stream. By making the stream long enough, adversary can force the competitive ratio to be greater than c.

Thus any algorithm with a non-trivial competitive ratio must keep the total weight of alive jobs low at all times. A natural strategy to consider is greedy, which at any time instant schedules the job with the highest weight to remaining processing time. However this is easily shown to be $\Omega(\sqrt{B})$ competitive [23]. The problem is that the algorithm does not distinguish between a job with a high weight and correspondingly high processing time and a small weight job with a small processing time (for example, a job of weight 1 and size 1, and another job of weight W and size W). Breaking ties badly, the naive greedy algorithm may be left with a single job with a large weight and a small remaining processing time, while the optimum algorithm only has the single job of small weight. On the other extreme, the algorithm which simply disregards the size of the job and works on the highest weight job is easily shown to be $B - competitive$ [23].

The $O(\log^2 B)$ competitive algorithm of [23] deals with the two problems above by striking a balance between them. It processes a large weight job without regard to its ratio as long as the total weight of jobs with a strictly better weight to remaining processing time ratio does not get too large.

However, even for the case when the job weights are either 1 or 2, the algorithm of [23] can be shown to be $\Omega(\log B)$ competitive[4]. Consider the following scenario: At time $t = 0$ two jobs, one of size B and weight 2 and other of size $B/2$ and weight 1 are released. For i ranging from 1 to $\log B - 1$, at each time instant $B(1 - 2^{-i})$, one job of weight 1 and size $B/2^{i+1}$ is released. Finally, at time $t = B(1 - 2^{i+1})$, another job of size $B/2^{i+1}$ is released.

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

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

3.4 Algorithm

We now present our algorithm that we call *Balanced SRPT*, for minimizing weighted flow time. We then analyze it and show that its analysis is tight. Finally we show that this algorithm can be modified to give an $O(\log n + \log B)$ competitive semi-online algorithm.

We assume that the weights of the jobs are drawn from the set $\{w_1, w_2, \ldots, w_k\}$, where $w_1 < w_2 \ldots < w_k$. We will refer to these as weight classes $1, \ldots, k$. Without loss of generality[5], we also assume that w_i/w_{i-1} is integral for $1 < i \leq k$. Let $Q_j(t)$ (or just Q_j when the time t is evident from the context) denote the set of alive jobs of class j at time t. For a collection of jobs X, we use $W(X)$ to mean the sum of the weights of the jobs in X.

Balanced SRPT: At all times, consider the class j, that has maximum $W(Q_j(t))$, breaking ties in favor of the higher weight class. Within Q_j execute the job with shortest remaining processing time.

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

3.4.1 Analysis

We now prove the k competitiveness of Balanced *SRPT*.

First we give notation for describing some useful quantities. We denote our Balanced *SRPT* algorithm by Bal and the optimum algorithm by Opt. The tuple (j, l) denotes job in weight class j that has the l^{th} largest remaining processing time among jobs in that class under Bal.

Define an order on 2-tuples of integers: We say that $(j, l) \prec (j', l')$ if and only if either of the following two conditions is satisfied.

[4] The algorithm of [23] rounds up the weights of incoming jobs to an integral power of $4(\log B + 1)$. Hence, it is trivially loses a $4(\log B + 1)$ factor in competitive ratio. However, our example does not use this rounding to prove $\Omega(\log B)$ competitiveness.

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

1. $w_j \cdot l < w_{j'} \cdot l'$

2. $w_j \cdot l = w_{j'} \cdot l'$ and $j < j'$

Also, we say $(j', l') \preceq (j, l)$ if $(j, l) \not\prec (j', l')$. Note that $(j', l') \preceq (j, l)$ and $(j, l) \preceq (j', l')$ holds iff $j = j'$ and $l = l'$, and that $\preceq$ defines a total order on the 2-tuples. Intuitively, the notion $\prec$ formalizes the order in which Bal executes the jobs. In particular, at any time Bal executes the job which has the highest priority according to the ordering $\prec$.

Let $B(j, l, t)$ be the set of jobs that are alive at time t under Bal and have priority no more than that of job (j, l). Thus, $B(j, l, t)$ consists of all jobs (j', l') such that $(j', l') \preceq (j, l)$ Let $|B(j, l, t)| = \sum_{J \in B(j,l,t)} rem(J, t)$, where $rem(J, t)$ is the remaining time of job J at time t. Figure 3.1 below describes the sets of jobs $B(j, l, t)$ for various of j and l.

The main proof idea will be to compare the "prefixes" $|B(j, l, t)|$ with suitable prefixes of Opt. We now define the prefix for Opt. Consider the jobs under Opt that are alive at time t, a valid packing $P(j, l, t)$ is a sub-collection of jobs alive under Opt at time t, such that the total weight of the jobs in $P(j, l, t)$ is at most $w_j \cdot l$. Let $|P(j, l, t)|$ denote the total remaining time of jobs contained in $P(j, l, t)$.

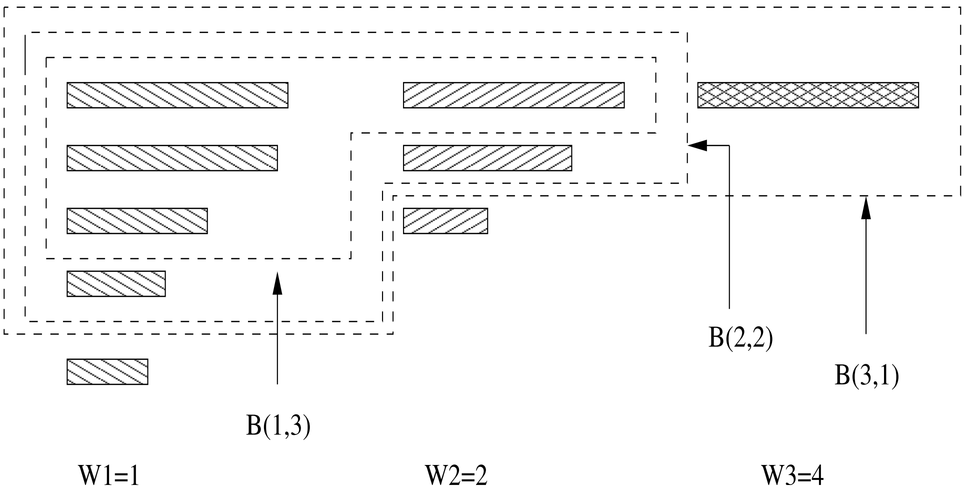


Figure 3.1: An example illustrating the notation used. We have dropped the third argument from B.

We begin by observing some properties of these prefixes.

Observation 3.1 *For $h \leq j$ and any l, $|B(j, l, t)| \geq |B(h, w_j/w_h \cdot l, t)|$.*

Proof. It follows directly from the definition of the $\prec$ relation that $B(h, w_j/w_h \cdot l, t) \subseteq B(j, l, t)$, for $h \leq j$, and hence $|B(h, w_j/w_h \cdot l, t)| \subseteq |B(j, l, t)|$. For example, in Figure 3.1 $B(1, 4, t) \subset B(2, 2, t)$. ■

Let $V(t)$ denote the total amount of work remaining at time under Bal (this is also equal to the work under any conserving policy and hence equal to the work under Opt at time t).

Observation 3.2 *At any time t, if Bal works on some job in $B(j, l, t)$, then $|B(j, l, t)| = V(t)$*

Proof. At any time t, Bal works on the job with the highest priority. Thus, if $B(j,l,t)$ decreases at time t, then the highest priority job must be in some position (j',l') such that $(j',l') \preceq (j,l)$. Thus, all the alive jobs lie in $B(j,l,t)$. ∎

Observation 3.3 *Suppose a job J of weight w_i and size $|J|$ arrives at time t, and that it is inserted in position h based on the remaining processing times among the weight w_i jobs under Bal. Let t^- denote the time just before J arrives, then*

1.For all j and l, $|B(j,l,t)| \geq |B(j,l,t^-)|$.

2.For all j and for all l such that $l > w_i/w_j$

$$|B(j,l,t)| \geq \begin{cases} |B(j,l-w_i/w_j,t^-)| + |J| & \text{if } i > j \\ |B(i,w_j \cdot l/w_i - 1,t^-)| + |J| & \text{if } i \leq j \end{cases}$$

Proof. Let f_i be the smallest number for which $(j,l) \prec (i,f_i)$. Let $|J'|$ denote the remaining processing time of the job at position (i,f_i-1) at time t^- ($|J'| = 0$ if there is no job at position (i,f_i-1)). Observe that if $l > w_i/w_j$, then for $i > j$, $(j,l-w_i/w_j) \prec (i,f_i-1) \prec (j,l)$ and hence

$$|B(j,l,t)| \geq |B(j,l-w_i/w_j,t)| + |J'| \tag{3.1}$$

Similarly, for $i \leq j$, we have that $f_i = w_j \cdot l/w_i + 1$ and hence $(i,w_j \cdot l/w_i - 1) \prec (i,f_i-1)$ and thus

$$|B(j,l,t)| \geq |B(i,l \cdot w_j/w_i,t) \geq |B(i,l \cdot w_j/w_i - 1,t)| + |J'| \tag{3.2}$$

We first show that $|B(j,l,t)| \geq |B(j,l,t^-)|$. We have two cases.

1.If $J \notin B(j,l,t)$, then clearly, $B(j,l,t) = B(j,l,t^-)$ and hence $|B(j,l,t)| = |B(j,l,t^-)|$.

2.In the case when $J \in B(j,l,t)$, we argue as follows. Since $J \in B(j,l,t)$ by our assumption, we must have that $h < f_i$. Then, the only way in which $B(j,l,t)$ differs from $B(j,l,t^-)$ is that J is inserted in position (i,h) and the jobs previously in positions $(i,h),\ldots,(i,f_i-1)$ move to positions $(i,h+1),\ldots,(i,f_i)$ respectively. Again, let $|J'|$ denote the remaining processing time of the job at position (i,f_i-1) at time t^-. Thus, we have that $|B(j,l,t)| = |B(j,l,t^-)| - |J'| + |J|$. Since, $h \leq f_i - 1$, it must be the case that $|J'| \leq |J|$. Thus, we have that $|B(j,l,t)| \geq |B(j,l,t^-)|$.

For the second part of the observation, we again have two cases.

1.First, if $J \notin B(j,l,t)$, then it must be the case that $|J| \leq |J'|$. Now it follows that, for $i > j$,

$$\begin{aligned} |B(j,l,t)| &\geq |B(j,l-w_i/w_j,t)| + |J'| \\ &\geq |B(j,l-w_i/w_j,t)| + |J| \\ &= |B(j,l-w_i/w_j,t^-)| + |J| \end{aligned}$$

Similarly, for $i \le j$,

$$\begin{aligned} |B(j,l,t)| &\ge |B(i, l \cdot w_j/w_i, t)| && \text{(by Observation 3.1)} \\ &\ge |B(i, l \cdot w_j/w_i - 1, t^-)| + |J'| && \text{(by Equation 3.2)} \\ &\ge |B(i, l \cdot w_j/w_i - 1, t^-)| + |J| && \text{(as } |J| \le |J'|\text{)} \end{aligned}$$

2.Next, if $J \in B(j,l,t)$, it must be that $|J| \ge |J'|$. Thus, we have that for $i > j$,

$$\begin{aligned} |B(j,l,t)| &\ge |B(j,l,t^-)| - |J'| + |J| \\ &\ge |B(j, l - w_i/w_j, t^-)| + |J| && \text{(by Equation 3.1)} \end{aligned}$$

Similarly, for $i \le j$,

$$\begin{aligned} |B(j,l,t)| &\ge |B(i, l \cdot w_j/w_i, t)| \\ &= |B(i, l \cdot w_j/w_i, t^-)| - |J'| + |J| \\ &\ge |B(i, l \cdot w_j/w_i - 1, t^-)| + |J| \end{aligned}$$

■

Recall that a valid *packing* $P(j,l,t)$ is any sub-collection of jobs alive under *Opt* at time t, such that the total weight of the jobs in $P(j,l,t)$ is at most $w_j \cdot l$.

Lemma 3.4 *At all times t, for all* $1 \le j \le k$ *and* $l \ge 1$, *and any packing* $P(j,l,t)$ *the following invariant holds*

$$|B(j,l,t)| \ge |P(j,l,t)| \tag{3.3}$$

Proof. We will prove Equation 3.3 by induction on the arrivals in the system. First, by perturbing the arrival times of jobs by an infinitesimal amount we can assume without loss of generality that all the arrival times are distinct. Let $0 = t_1 \le t_2 \le \ldots \le t_n$ denote the arrival times of the n jobs. Clearly, the invariant is true at $t_1 = 0$.

Suppose the invariant is true at some t_i. We then show that the invariant 3.3 is true at all times in the interval $[t_i, t_{i+1})$. For the sake of contradiction, let $t \in [t_i, t_{i+1})$ be the first time when the invariant 3.3 fails to hold. Let $t' \in (t_i, t]$ be the last time when $B(j,l,t')$ was decreasing.

If $t = t'$, we know that $|B(j,l,t)| = V(t)$, and hence $|B(j,l,t)| \ge P(j,l,t)|$.

If $t' < t$, and since $|B(j,l,t)|$ did not decrease during $(t,t']$, and as there were no arrivals during this time, we know that $|B(j,l,t)| = |B(j,l,t')| = V(t')$. Moreover, we also have that for any collection of jobs $P(j,l,t)$, $|P(j,l,t)| \le |P(j,l,t')|$, because *Opt* could have worked on some of these jobs during $[t',t]$. Since $|P(j,l,t')| \le V(t')$, if $|B(j,l,t)| < |P(j,l,t)|$, then we have that $|B(j,l,t')| = |B(j,l,t)| < |P(j,l,t)| \le |P(j,l,t')|$, contradicting the fact that t was the earliest time when the invariant 3.3 fails to hold.

We now consider the case of job arrivals. Let t^- denote the time instance just before time t. Suppose there is an arrival at time t and that the invariant 3.3 fails to hold at time t then we show that it could not be true for some packing at time t^-.

Let $P(j,l,t)$ denote the packing for which $|B(j,l,t)| < |P(j,l,t)|$ and let J denote the job that arrived at time t.

First we note that if $J \notin P(j,l,t)$ then $P(j,l,t)$ is also a valid packing at time t^-, and that $|P(j,l,t)| = |P(j,l,t^-)|$. Also, by Observation 3.3 $|B(j,l,t)| \geq |B(j,l,t^-)|$, hence if $|B(j,l,t)| < |P(j,l,t)|$, then this would contradict the fact that $|B(j,l,t^-)| \geq |P(j,l,t^-)|$.

We now consider the case, when the new job is included in the packing $P(j,l,t)$. We i denote the class of the job which arrives at time t. We consider two cases depending on whether $i > j$ or $i \leq j$.

$i > j$: By Observation 3.3 we know that, $B(j,l,t) \geq B(j,l-w_i/w_j,t^-) + |J|$. Consider the packing $P'(j,l-w_i/w_j,t^-)$ at time t^- which is obtained from $P(j,l,t)$ by removing the job J. Then we have that $|P(j,l,t)| = |P'(j,l-w_i/w_j,t^-)| + |J|$. Thus, if $B(j,l,t) < P(j,l,t)$, then this would contradict the fact that $|B(j,l-w_i/w_j,t^-)| \geq |P'(j,l-w_i/w_j,t^-)|$.

$i \leq j$: By Observation 3.3 we know that, $B(j,l,t) \geq B(i,w_j/w_il-1,t^-)| + |J|$. Again, we consider the packing $P'(i,w_jl/w_i-1,t^-)$ at time t' which is obtained from $P(j,l,t)$ by removing the job J. Then we have that $|P(j,l,t)| = |P'(j,l-w_i/w_j,t^-)| + |J|$. Thus, if $B(j,l,t) < P(j,l,t)$, then this would contradict the fact that $|B(i,w_jl/w_i-1,t^-)| \geq |P'(j,w_jl/w_i-1,t^-)|$.

■

Theorem 3.5 *The Balanced SRPT algorithm is k competitive for k weight classes.*

Proof. For any arbitrary time t, let $opt(t)$ denote the total weight of alive jobs under the optimum algorithm. Clearly, $opt(t)$ is a multiple of w_1, since w_i/w_{i-1} is an integer for each $1 < i \leq k$. Choosing $j = 1$ and $l = opt(t)/w_1$ and considering the packing $P(j,l,t)$ which contains all the jobs present under the optimum algorithm. We get that $|P(j,l,t)| = V(t)$, where $V(t)$ is the total volume of jobs at time t. By Lemma 3.3, we have that $|B(j,l,t)| \geq |P(j,l,t)| = V(t)$. However, since Bal is work-conserving we have that $|B(j,l,t)| \leq V(t)$ and hence $|B(j,l,t)| = V(t)$. Thus, $B(j,l,t)$ contains all the jobs, and since $wt(B(j,l,t)) \leq k \cdot w_j \cdot l = k \cdot opt(t)$, this implies the k competitiveness. ■

Analysis of Balanced *SRPT* is tight

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

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

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

3.4.2 A Semi-Online Algorithm

Given an instance I, assume without loss of generality that the job sizes are integers and that the smallest job weight is 1. Suppose we know the values of the largest weight, largest job size and the total number of jobs (i.e. W, B and n). Then we can modify the Balanced *SRPT* algorithm in the following way: Round the weights up to the nearest power of 2. Consider only the jobs with weights between $\frac{W}{Bn^2}$ and W. Call these jobs *heavy*. Run the Balanced *SRPT* algorithm on the heavy jobs, if at some time no heavy job is present, work on any arbitrary job.

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

When restricted to heavy jobs, by Theorem 3.5, the performance of the Balanced *SRPT* is no worse than $4(\log n + \log B)$ times the optimum (restricted to I_h). Thus,

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

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

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

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

3.5 Weighted ℓ_p norms of Flow Time

We now consider the problem of minimizing the weighted ℓ_p norms of flow time. We first define some needed notation. For an algorithm A on an input instance $\mathcal{I}$ with an s speed processor, let $F^p(A, \mathcal{I}, s)$ denote the sum of the p^{th} powers of the flow time of all jobs. Similarly, $WF^p(A, \mathcal{I}, s)$ will denote the sum of weighted p^{th} powers of the flow time (i.e. $\sum_i w_i f_i^p$) of all jobs. Finally, for the measure F^p, let $Opt(F^p, \mathcal{I}, s)$ denote the value of the optimum schedule for the F^p measure on $\mathcal{I}$ with a speed s processor. Let $Opt(WF^p, \mathcal{I}, s)$ denote the optimum value for the WF^p measure.

We now define the general transformation that we will use throughout this section.

Transforming a weighted to an unweighted instance: Given an instance $\mathcal{I}$, we define an instance $\mathcal{I}'$ as follows:
Consider a job $J_i \in \mathcal{I}$. The instance $\mathcal{I}'$ is obtained by replacing J_i by w_i identical jobs each of size p_i/w_i and weight 1, and release time r_i. We denote these w_i jobs by $J'_{i1}, \ldots, J'_{iw_i}$. Let $X_i = \{J'_{i1}, \ldots, J'_{iw_i}\}$ denote this collection of jobs obtained from J_i. Note that all jobs in $\mathcal{I}'$ have the same weight.

3.5.1 The Clairvoyant Case

In this section we show that *HDF*, a natural generalization of *SJF* is a $(1+\epsilon)$-speed, $O(1/\epsilon^2)$-competitive online algorithm for minimizing the weighted ℓ_p norms of flow time.

The main idea of the proof will be to reduce the weighted problem to an unweighted problem using the transformation described above and then invoke the result for ℓ_p norms of unweighted flow time.

Lemma 3.6 *For $\mathcal{I}$ and $\mathcal{I}'$ as defined above,*

$$Opt(F^p, \mathcal{I}', 1) \leq Opt(WF^p, \mathcal{I}, 1) \tag{3.4}$$

Proof. Let S be the schedule which minimizes the weighted ℓ_p norm of flow time for $\mathcal{I}$. Given S, we create a schedule for $\mathcal{I}'$ as follows. At any time t, work on a job in X_i if and only if J_i is executed at time t under S. Clearly, all jobs in X_i finish when J_i finishes execution, thus no job in X_i has a flow time higher than that of J_i. By definition, the contribution of J_i to WF^p is $w_i f_i^p$. Also, the contribution to the measure F^p of each of the w_i jobs in X_i will be at most f_i^p, and hence the total contribution of jobs in X_i to F^p is at most $w_i f_i^p$. Since the optimum schedule for $\mathcal{I}'$ can be no worse than the schedule constructed above, the result follows. ■

From Theorem 2.3 in Chapter 2 we know that *SJF* is $(1+\epsilon)$-speed, $O(1/\epsilon)$-competitive for the (unweighted) ℓ_p norms of flow time , or equivalently *SJF* is $(1+\epsilon)$-speed, $O(1/\epsilon^p)$-competitive for the F^p measure. This implies that,

$$F^p(\textit{SJF}, \mathcal{I}', 1+\epsilon) = O(\frac{1}{\epsilon^p}) Opt(F^p, \mathcal{I}', 1) \tag{3.5}$$

We now relate the performance of *HDF* on $\mathcal{I}$ with a $(1+\epsilon)$ times faster processor to that of *SJF* on $\mathcal{I}'$. Recall that *HDF* is the algorithm that any time works on the job with the small ratio of service received to its weight.

Lemma 3.7

$$WF^p(\textit{HDF}, \mathcal{I}, 1+\epsilon) \leq (1+\frac{1}{\epsilon})^p F^p(\textit{SJF}, \mathcal{I}', 1) \tag{3.6}$$

Proof. We claim that for every job $J_i \in \mathcal{I}$ and every time t, if J_i is alive at time t under *HDF* with a $(1+\epsilon)$ speed processor, then at least $\frac{\epsilon}{1+\epsilon} w_i$ jobs in $X_i \in \mathcal{I}'$ are alive at time t under *SJF* with a 1 speed processor.

The claim above immediately implies the result for the following reason. Consider the time $t^- = (f_i + r_i)^-$ just before J_i finishes execution under *HDF*. Then J_i contributes exactly $w_i f_i^p$ to $WF^p(\textit{HDF}, \mathcal{I}, 1+\epsilon)$, while the $\geq \epsilon w_i/(1+\epsilon)$ jobs in X_i that are unfinished by time t contribute at least $\epsilon w_i/(1+\epsilon) f_i^p$ to $F^p(\textit{SJF}, \mathcal{I}', 1)$. Taking the contribution over each job, the result follows.

We now prove the claim. Suppose for the sake of contradiction that t is the earliest time when J_i is alive under *HDF* and there are fewer than $\epsilon/(1+\epsilon) w_i$ jobs from X_i left under *SJF*. Since J_i is alive under *HDF* and *HDF* has a $(1+\epsilon)$ faster processor, it has spent less than $p_i/(1+\epsilon)$ time on J_i, whereas *SJF* has spent strictly more than $p_i/(1+\epsilon)$ time on X_i. Thus there was a some time t', such that $r_i \leq t' < t$ during which *HDF* was running $J_j \neq J_i$ while *SJF* was working on some job from X_i. Since $t' \geq r_i$, it follows from the property of *HDF* that J_j has higher density than that of J_i. This implies that jobs in X_j have smaller size than X_i. Since *SJF* works on X_i at time t', it must have already finished all the jobs in X_j by t'. Since J_j is alive at time t', this contradicts our assumption of the minimality of t. ∎

By Equations 3.5 and 3.6 we have that

$$WF^p(\mathit{HDF}, \mathcal{I}, (1+\epsilon)^2) = O(1/\epsilon)^{2p} Opt(F^p, \mathcal{I}', 1)$$

Combining this with Equation 3.4 we obtain that,

Theorem 3.8 *HDF is $(1+\epsilon)$-speed, $O(1/\epsilon^2)$-competitive for minimizing the weighted ℓ_p norms of flow time.*

3.5.2 The Non-Clairvoyant Case

As in the analysis of *HDF* the main step of our analysis will be to relate the behavior of *WSETF* on an instance $\mathcal{I}$ with weighted jobs to that of *SETF* on another instance $\mathcal{I}'$ which consists of unweighted jobs. We then use the results about (unweighted) ℓ_p norms of flow time under *SETF* to obtain results for *WSETF*.

Given an instance $\mathcal{I}$ consisting of weighted jobs, let $\mathcal{I}'$ denote the instance defined as in Section 3.5.1 which consists of unweighted jobs. Suppose we run *WSETF* on $\mathcal{I}$ and *SETF* on $\mathcal{I}'$ with the same speed processor. Then the schedules produced by *WSETF* and *SETF* are related by the following simple observation.

Lemma 3.9 *At any time t, a job $J_i \in \mathcal{I}$ is alive and has received $p_i(t)$ units of service if and only if each job in $X_i \in \mathcal{I}'$ is alive and has received exactly $p_i(t)/w_i$ amount of service. In particular, this implies that if J_i has flow time f_i then each $J'_{ik} \in X_i$ for $k = 1, \ldots, w_i$ has flow time f_i.*

Proof. We view the execution of *WSETF* on $\mathcal{I}$ as follows: If at any time *WSETF* allocates x units of processing to a job of weight w_i, then we think of it as allocating x/w_i units of processing to each of the w_i jobs in the collection X_i. Thus the normalized service of job J_i under *WSETF* is exactly equal to the amount of service received by a job in X_i. Since *WSETF* at any time shares the processor among jobs with the smallest normalized service in the ratio of their weights, this is identical to the behavior of *SETF* on $\mathcal{I}'$ which works equally on the jobs which have received the smallest amount of service. ∎

Theorem 3.10 *WSETF is a $(1+\epsilon)$-speed, $O(1/\epsilon^{2+2/p})$-competitive non-clairvoyant algorithm for minimizing the weighted ℓ_p norms of flow time.*

Proof. By Lemma 3.9 we know that if $J_i \in \mathcal{I}$ has flow time f_i, then the w_i jobs in X_i have flow time f_i. Thus the ℓ_p norm of unweighted flow time for $\mathcal{I}'$ is $(\sum_i w_i f_i^p)^{1/p}$ which is identical to the weighted flow time for $\mathcal{I}$ under *WSETF*, this implies that

$$WF^p(\mathit{WSETF}, \mathcal{I}, 1) = F^p(\mathit{SETF}, \mathcal{I}', 1) \tag{3.7}$$

By Lemma 3.4 we know that $Opt(F^p, \mathcal{I}', 1) \leq Opt(WF^p, \mathcal{I}, 1)$. By the main result of Section 7 in [11] about the competitiveness of *SETF* for unweighted ℓ_p norms of flow time we know that

$$F^p(\mathit{SETF}, \mathcal{I}', (1+\epsilon)) = O(1/\epsilon^{2p+2}) Opt(F^p, \mathcal{I}', 1) \tag{3.8}$$

Now, by Equations 3.7, 3.8 and 3.4 we get that

$$WF^p(\textit{WSETF}, \mathcal{I}, 1+\epsilon) = O(1/\epsilon^{2p+2})Opt(WF^p, \mathcal{I}, 1)$$

Thus the result follows. ∎

Improved bounds for $p = 1$

Theorem 3.10 implies that for $p = 1$, *WSETF* is a $(1 + \epsilon)$-speed, $O(1/\epsilon^4)$ competitive algorithm. We now show that *WSETF* is in fact $(1 + \epsilon)$-speed, $O(1/\epsilon)$ competitive for $p = 1$.

To see this, we simply use the fact that $SETF$ is $(1 + \epsilon)$-speed, $O(1/\epsilon)$-competitive for minimizing total unweighted flow time [46]. Combining this with Equation 3.7 we have that

$$WF^1((\textit{WSETF}, \mathcal{I}, 1+\epsilon) = F^1(\textit{SETF}, \mathcal{I}', 1+\epsilon) \leq \frac{1}{\epsilon}Opt(F^1, \mathcal{I}', 1)$$

Finally, since $Opt(F^p, \mathcal{I}', 1) \leq Opt(WF^p, \mathcal{I}, 1)$ (by Lemma 3.4) the result follows.

3.6 Conclusion and Open Problems

In this chapter we gave an $O(\log W)$ competitive online algorithm for minimizing the total weighted flow time on a single machine with preemptions. We also saw how resource augmentation results for weighted flow time can often be obtained directly using results for the unweighted case.

An outstanding open problem is the following:

Open Problem 3.3 Is there an $O(1)$ competitive online or a polynomial time $O(1)$ approximation algorithm for minimizing the weighted flow time on a single machine with preemptions?

Chapter 4

Scheduling to Minimize Stretch

4.1 Introduction

In this chapter we study problems related to minimizing stretch on a single machine with preemptions. We will consider two problems:

1. Minimizing the total stretch.
2. Minimizing the ℓ_p norms of stretch.

Even though the first problem is a special case of the second (with $p = 1$), we discuss them separately because the case of $p = 1$ is special and has also been studied prior to our work. Also, the first problem is easier from the second in the sense: While a 2-competitive clairvoyant online algorithm is known for minimizing the total stretch [60], however, as we shall see, there cannot be any online clairvoyant algorithm that is $n^{o(1)}$ competitive for minimizing the ℓ_p norms of stretch for $p > 1$.

Stretch is a special case of weighted flow time where the weight of a job is inversely proportional to its size. Hence, a $(1 + \epsilon)$-speed, $O(1)$-competitive algorithm for minimizing the ℓ_p norms of stretch in the clairvoyant setting follows directly from the more general results on minimizing the weighted ℓ_p norms of flowtime (Theorem 3.8 in Chapter 3).

Minimizing stretch becomes more interesting the non-clairvoyant case. In this setting, the problem is considerably harder than the weighted flow time problem. 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). This is reflected in the strong lower bounds for this problem. We will show that any algorithm is $\Omega(n)$ and $\Omega(B)$ competitive for minimizing total stretch. Even with randomization and a factor k speed up the stretch problem is at least $\Omega(n/k)$ competitive. Note especially that neither resource augmentation nor randomization seems to help. This is in sharp contrast to weighted flow time where a $(1 + \epsilon)$-speed gives a $(1 + \frac{1}{\epsilon})$-competitive algorithm [7]. 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 [45, 13]. The only case where resource augmentation seems to help is when the job sizes are bounded. Our main result is that,

Theorem 4.1 *SETF is a $(1 + \epsilon)$-speed, $O(\frac{1}{\epsilon^{3+1/p}} \cdot \log^{1+1/p} B)$-competitive algorithm for the ℓ_p norm of stretch.*

This result is almost tight, as also show that any randomized non-clairvoyant algorithm with an $O(1)$ speed is $\Omega(\log B)$ competitive for any ℓ_p norm of stretch.

For the case of $p = 1$, Theorem 4.1 implies that *SETF* is a $(1+\epsilon)$-speed, $O(\log^2 B)$-competitive algorithm for minimizing total stretch non-clairvoyantly. This is currently the best known upper bound for this problem. The $(1+\epsilon)$-speed, $O(\log^2 B)$ competitiveness of SETF for $p = 1$ was first shown in [8]. The analysis technique from [8] was later simplified and extended in [11] for the case of general $p \geq 1$. The table 4.1 summarizes our results for non-clairvoyant scheduling.

Problem Scenario	Speed	*SETF*	Lower bound without speed up	Lower bound with $O(1)$ speed up
Static	1	$O(n), O(\log B)$	$\Omega(n), \Omega(\log B)$	$\Omega(n), \Omega(\log B)$
Dynamic	$(1+\epsilon)$	$O(\log^{1+1/p} B)$	$\Omega(n), \Omega(B)$	$\Omega(n), \Omega(\log B)$

Table 4.1: Summary of results for minimizing the ℓ_p norms of stretch non-clairvoyantly.

This chapter is organized as follows: In Section 4.2 we discuss the previous work. In Section 4.3 we show various lower bounds. We first show that there cannot exist any $n^{o(1)}$ competitive clairvoyant online algorithm for minimizing the ℓ_p norms of stretch for $p > 1$. Later we consider the lower bounds for non-clairvoyant algorithms. Next, we give the positive results. In Section 4.4 consider the static case, (i.e., when all requests arrive at time $t = 0$) and show that SETF is the optimal algorithm up to constant factors. In Section 4.5 we consider the dynamic case and prove Theorem 4.1. [1]

4.2 Related Work

Stretch of a job was first considered by Bender *et al.* [17]. Stretch as a metric has received much attention lately [27, 75, 60, 16, 15, 6, 18], since it captures the notion of "fairness". Note that a low stretch implies that jobs are not delayed disproportionately to their size, hence smaller jobs are delayed less and large jobs are delayed proportionately more. Muthukrishnan *et al.* [60] first studied total stretch, 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. In the offline case, there a PTAS is known for minimizing total stretch on a single machine [18]. Recently, there have been various extensions and improvements [23, 6, 15, 18] to the problem. However, all previous work looks at the problem in the clairvoyant setting. Since clairvoyant scheduling does not accurately model many systems, there is significant interest in the non-clairvoyant version of stretch.

4.3 Lower Bounds

4.3.1 Clairvoyant Scheduling

Theorem 4.2 *The competitive ratio of any randomized algorithm A against an oblivious adversary for F^p, $1 < p < \infty$, is $n^{\Omega(1)}$. In particular, the competitive ratio of any randomized algorithm is $\Omega(n^{(p-1)/3p^2})$.*

[1]The results in this chapter are combined from [8, 11, 7, 10].

Proof. We use Yao's minimax principle for online cost minimization problems [19] and lower bound the expected value of the ratio of the objective functions of A and *Opt* on input distribution which we specify. The inputs are parameterized by integers L, α, and β in the following manner. A *long* job of size L arrives at $t = 0$. From time 0 until time $L^\alpha - 1$ a job of size 1 arrives every unit of time. With probability $1/2$ this is all of the input. With probability $1/2$, $L^{\alpha+\beta}$ *short* jobs of length $1/L^\beta$ arrive every $1/L^\beta$ time units from time L^α until $2L^\alpha - 1/L^\beta$.

We now consider the S^p objective. In this case, $\alpha = \frac{2p+1}{p-1}$, and $\beta = 1$. We now compute $S^p(A)$ and $S^p(\mathit{Opt})$. Consider first the case that A doesn't finish the long job by time L^α. Then with probability $1/2$ the input contains no short jobs. Then $S^p(A)$ is at least the stretch of the long job, which is at least $(L^\alpha/L)^p = L^{p(\alpha-1)}$. In this case the adversary could first process the long jobs and then process the unit jobs. Hence, $S^p(\mathit{Opt}) = O(1 + L^\alpha \cdot L^p) = O(L^{\alpha+p})$. The competitive ratio is $\Omega(L^{\alpha p-\alpha-2p})$, which is $\Omega(L)$ by our choice of α.

Now consider the case that A finishes the long job by time L^α. Then with probability $1/2$ the input contains short jobs. One strategy for the adversary is to finish all jobs, except for the big job, when they are released. Then $S^p(\mathit{Opt}) = O(L^\alpha \cdot 1^p + L^{\alpha+\beta} \cdot 1^p + (L^\alpha/L)^p)$. Algebraic simplification shows that $\alpha + \beta \leq \alpha p - p$ for our choice of α and β. Hence dominant term is $L^{\alpha p - p}$, and $S^p(\mathit{Opt}) = O(L^{\alpha p-p})$. Now consider the subcase that A has at least $L/2$ unit jobs unfinished at time $3L^\alpha/2$. Since these unfinished unit jobs must have been delayed by at least $L^\alpha/2$, $S^p(A) = \Omega(L \cdot L^{\alpha p})$. Clearly in this subcase the competitive ratio is $\Omega(L^{p+1})$. Alternatively, consider the subcase that A has finished at least $L/2$ unit jobs by time $3L^\alpha/2$. Then A has at least $L^{1+\beta}/2$ released, and unfinished, small jobs at time $3L^\alpha/2$. By the convexity of S^p when restricted to jobs of size $1/L^\beta$, the optimal strategy for A from time $3L^\alpha/2$ onwards always delays each small job by the same amount (we can gift A the competition of the unit jobs at time $3L^\alpha/2$). Thus A delays $L^{\alpha+\beta}/2$ short jobs by at least $L/2$. Hence in this case, $S^p(A) = \Omega(L^{\alpha+\beta} \cdot L^{(\beta+1)p})$. This gives a competitive ratio of $S^p(A) = \Omega(L^{\alpha+\beta+\beta p+2p-\alpha p})$. By the choice of α and β, this gives a competitive ratio of $\Omega(L^{2p+1})$. Using that $n = O(L^{\alpha+\beta})$ we obtain the desired lower bound. ∎

We note that Theorem 4.2 is not particularly tight. In particular, as p approaches infinity Theorem 4.2 does not yield any substantial lower bounds, however for $p = \infty$, a lower bound of $\Omega(n^{1/2})$ on the competitive ratio of any deterministic online algorithm was shown by Bender et al [17].

4.3.2 Non-clairvoyant Scheduling

We now give lower bounds for the non-clairvoyant case. These lower bounds suggest that the only meaningful parameter for which interesting results can be obtained is B, the bound on job sizes.

Bounded job sizes without resource augmentation

We show that any non-clairvoyant algorithm *without speed up*, deterministic or randomized is at least $\Omega(B)$ competitive for minimizing total stretch (hence it is $\Omega(B)$ competitive for general ℓ_p norms of stretch).

Theorem 4.3 *Any non-clairvoyant algorithm, deterministic or randomized is at least $\Omega(B)$ competitive for total stretch, 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 stretch 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 stretch 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. ∎

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 4.3, the scheduling instance being static implies that even resource augmentation by k times can only help by a factor of k.

Lemma 4.4 *For minimizing the total stretch, 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 *SJF* behaves on this problem instance. The total stretch 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 *SJF* 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 stretch to be $\Omega(\log^2 B)$. The rough idea is as follows: We order the jobs of our instance such that for all $0 \le i \le \log B$, 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 stretch of A on the given instance is

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

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 total stretch 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 stretch is $\frac{1}{2}(\log B - i + 1)$ and the total stretch is $\Omega(\log^2 B)$. We now use Yao's Minimax Lemma to obtain the result. ■

As the input instance in Lemma 4.4 is static, a k-speed processor can at most improve all the flow times by a factor of k. Hence the total stretch can go down by the same factor. This gives us the following theorem.

Theorem 4.5 *Any k-speed deterministic or randomized, non-clairvoyant algorithm has an $\Omega(\log B/k)$ competitive ratio for minimizing total stretch, in the static case (and hence in the online case).*

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 4.6 *Any k-speed non-clairvoyant algorithm, either deterministic or randomized, has $\Omega(n/k)$ performance ratio for minimizing total stretch.*

4.4 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 *SETF* is $O(\log B)$ competitive for all ℓ_p norms of stretch, hence matching the lower bound shown in the previous section. We will prove this in two steps: We first show that S^p for the optimum algorithm on any input instance with n jobs is $\Omega(n^{p+1}/\log^p B)$. Then we show that S^p for *SETF* on any input instance with n jobs is $O(n^{p+1})$.

In the static case, it is easily seen by a simple exchange argument that *SJF* is the optimal algorithm for ℓ_p norms of stretch for every $p \geq 1$. We now show a general lower bound on the ℓ_p norms of stretch under *SJF* on any input instance.

Lemma 4.7 *For* any *scheduling instance with n jobs all arriving at time 0, SJF has S^p at least $\Omega(n^{p+1}/\log^p B)$.*

Proof. We first show that without loss of generality 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 stretch under *SJF* as follows: For a job of size x, we require *SJF* 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 the stretch. Thus, we can round down all the job sizes to a power of 2 and have the new stretch of each job within a factor of 4 of its stretch in original instance.

Given an instance $\mathcal{I}$ consisting of n jobs. Let $x_1, x_2 \ldots, x_{\log B}$ denote the number of jobs of sizes $2, 4 \ldots, B$ respectively. We also have that $\sum x_i = n$. Now, as there are at least x_i jobs of size 2^i, the sum of the p^{th} powers of flow time of the jobs of size 2^i under *SJF* is at least $\sum_{k=1}^{x_i} k^p 2^{ip} \geq$

$x_i^{p+1} 2^{ip}/(p+1)$. Thus the sum of the p^{th} powers of stretch of such jobs is at least $x_i^{p+1}/(p+1)$. Thus summing up over all job sizes the sum of the p^{th} powers of stretch is at least $\sum_i x_i^{p+1}/(p+1)$. By convexity of the function $f(x) = x^{p+1}$ for $p \geq 0$, this is at least $\log B (\sum_i x_i / \log B)^{p+1}$ which is $\Omega(n^{p+1}/\log^p B)$. ■

Lemma 4.8 *For a scheduling instance with n jobs, the ℓ_p norms of stretch under SETF is $O(n^{1+1/p})$.*

Proof. 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. Thus the stretch of any job is at most n and hence the sum of the p^{th} power of stretch of all the jobs is at most n^{p+1} which implies the result. ■

Combining the results of Lemmas 4.7 and 4.8, we get the following result:

Theorem 4.9 *For static scheduling with bounded job sizes, the SETF algorithm is $O(\log B)$-competitive for minimizing the ℓ_p norms of stretch.*

4.5 Dynamic Scheduling

We now prove Theorem 4.1. The analysis technique is similar to the analysis for flow time in Chapter 2, Section 2.6.

We first compare *SETF* to an intermediate policy *MLF*, and then compare *MLF* to *Opt*. The variant of MLF that we use is similar to that in Section 2.6. We describe it again below for completeness. Our *MLF* is a variation of the standard Multilevel Feedback Queue algorithm where the quanta for the queues are set as a function of ϵ. Let $\ell_i = \epsilon((1+\epsilon)^i - 1)$, $i \geq 0$, and let $q_i = \ell_{i+1} - \ell_i = \epsilon^2(1+\epsilon)^i$, $i \geq i$. In *MLF* a job J_j is in *level k at time t* if the work done on J_j by time t is $\geq \ell_k$ and $< \ell_{k+1}$. *MLF* maintains the invariant that it is always running the earliest arriving job in the smallest nonempty level.

Let *SETF*$_s$ (resp. *MLF*$_s$) denote the algorithms *SETF* and *MLF* executed with an s speed processor. By Lemma 2.6, it follows that

$$S^p(\textit{SETF}(\mathcal{I}), 1+\epsilon) \leq S^p(\textit{MLF}(\mathcal{I}), 1) \tag{4.1}$$

We now transform the input instance. Let the original instance be $\mathcal{I}$. Let $\mathcal{J}$ be the instance obtained from $\mathcal{I}$ as follows. Consider a job J_j and let i be the smallest integer such that $p_j + \epsilon \leq \epsilon(1+\epsilon)^i$. The processing time of J_j in $\mathcal{J}$ is then $\epsilon(1+\epsilon)^i$. Let $\mathcal{K}$ be the instance obtained from $\mathcal{J}$ by decreasing each job size by ϵ. Thus, each job in $\mathcal{K}$ has size $\ell_k = \epsilon((1+\epsilon)^k - 1)$ for some k. Note that in this transformation from $\mathcal{I}$ to $\mathcal{K}$, the size of a job doesn't decrease, and it increases by at most a factor of $(1+\epsilon)^2$. Since *MLF* is has the property that increasing the length of a particular job will not decrease the completion time of any job, we can conclude that

$$S^p(\textit{MLF}(\mathcal{I}), 1) \leq (1+\epsilon)^{2p} S^p(\textit{MLF}(\mathcal{K}), 1) \tag{4.2}$$

Equations 4.1 and 4.2 together give us that

$$S^p(\textit{SETF}(\mathcal{I}), 1+\epsilon) \leq (1+\epsilon)^{2p} S^p(\textit{MLF}(\mathcal{K}), 1) \tag{4.3}$$

We now create an instance $\mathcal{L}$ by replacing each job of size $\epsilon((1+\epsilon)^k - 1)$ job in $\mathcal{K}$ by k jobs of size $q_1, \ldots, q_{k-1}$. Note that $\ell_k = q_0 + q_1 + \ldots, q_{k-1}$. For a job of $J_j \in \mathcal{K}$, we denote the corresponding jobs in $\mathcal{L}$ by $J_{j_0}, J_{j_1}, \ldots, J_{j_{k-1}}$. Notice that any time t, *SJF*($\mathcal{L}$) is working on a job $J_{j_b} \in \mathcal{L}$ if and only if *MLF*($\mathcal{K}$) is working on job $J_j \in \mathcal{K}$ that is in level b at time t. In particular, this implies that the completion time of J_j in *MLF*($\mathcal{K}$) is exactly the completion time of some job $J_{j_b} \in$ *SJF*($\mathcal{L}$). Hence,

$$F^p(\mathit{MLF}(\mathcal{K}), s/(1+\epsilon)) \leq F^p(\mathit{SJF}(\mathcal{L}), s/(1+\epsilon)) \tag{4.4}$$

For a job of size $\epsilon[(1+\epsilon)^k - 1]$ in $\mathcal{K}$, the corresponding job of size $\epsilon^2(1+\epsilon)^k \in \mathcal{L}$ has equal flow time. Thus the ratio of the contributions to the p^{th} power of stretch for $\mathcal{L}$ and $\mathcal{K}$ will be at least $(\frac{\epsilon^2(1+\epsilon)^k}{\epsilon[(1+\epsilon)^k-1]})^p$, which is at least $(\frac{\epsilon}{1-(1+\epsilon)^{-k}})^p$. Now since $\epsilon[(1+\epsilon)^k - 1] \geq 1$ for all valid job sizes in $\mathcal{K}$, we get that $(1+\epsilon)^{-k} \leq \frac{\epsilon}{1+\epsilon}$ and hence that $(\frac{\epsilon}{1-(1+\epsilon)^{-k}})^p \leq (\epsilon(1+\epsilon))^p$. Thus we get that

$$S^p(\mathit{MLF}(\mathcal{K}), 1) \leq (\epsilon(1+\epsilon))^p S^p(\mathit{SJF}(\mathcal{L}), 1)$$

Now, applying Theorem 2.3 it follows that $S^p(\mathit{SJF}(\mathcal{L}), 1+\epsilon) = O(1/\epsilon^p) S^p(\mathit{Opt}(\mathcal{L}), 1)$. Thus we get that

$$S^p(\mathit{MLF}(\mathcal{K}), 1) = O((1+\epsilon)^p) S^p(\mathit{Opt}(\mathcal{L}), 1) \tag{4.5}$$

Let $\mathcal{L}(k)$ denote the instance obtained from $\mathcal{J}$ by multiplying each job size in $\mathcal{J}$ by $\epsilon/(1+\epsilon)^k$. Next, we remove from $\mathcal{L}(k)$ any job whose size is less than ϵ^2. We claim that $\mathcal{L} = \mathcal{L}(1) \cup \mathcal{L}(2) \cup \ldots$. To see this, let us consider some job $J_j \in \mathcal{J}$ of size $\epsilon(1+\epsilon)^i$. Then, $L(1)$ 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 $\mathcal{L}(1) \cup \mathcal{L}(2) \cup \ldots$.

Lemma 4.10 *Let $\mathcal{L}(k)$ be as defined above. $S(s, J_i, \mathcal{G})$ denote the stretch of job $J_i \in \mathcal{G}$ when SJF is run on $\mathcal{G}$ with a speed s processor. Then, for all $x \geq 1$,*

$$S(\epsilon(1+\epsilon)^{-k} \cdot x \cdot s, J_{i_k}, \mathcal{L}(k) \leq \frac{(1+\epsilon)^k}{\epsilon x} S(s, J_i, \mathcal{J})$$

Proof. Let $F(s, J_i, \mathcal{G}))$ denote the flow time of job $J_i \in \mathcal{G}$ when *SJF* is run on $\mathcal{G}$ with a speed s processor. By Lemma 2.7, we have that

$$F(\epsilon(1+\epsilon)^{-k} \cdot x \cdot s, J_{i_k}, \mathcal{L}(k)) \leq \frac{1}{x} F(s, J_i, \mathcal{J})$$

Since the size of jobs in $\mathcal{L}(k)$ are $\epsilon(1+\epsilon)^{-k}$ times smaller than in $\mathcal{J}$, the result follows. ■

We now prove the crucial result (similar to Lemma 2.8) that connects the performance of the optimum algorithm on $\mathcal{L}$ to the performance of *SJF* on $\mathcal{J}$.

Lemma 4.11

$$S^p(\mathit{Opt}(\mathcal{L}), 1+\epsilon) = O(\frac{\log_{1+\epsilon}^{p+1} B}{\epsilon^p}) S^p(\mathit{SJF}(\mathcal{J}), 1)$$

Proof. We will construct a schedule for $\mathcal{L}$ using the *SJF* schedule for $\mathcal{J}$.

Set $x_i = \epsilon(1+\epsilon)^{-i}$ for $i = 1, \ldots, \log_{1+\epsilon}\log_{1+\epsilon}(B/\epsilon)$, and $x_i = \epsilon/\log_{1+\epsilon}(B/\epsilon)$ for $i > \log_{1+\epsilon}\log_{1+\epsilon}(B/\epsilon)$.

We run the jobs in $\mathcal{L}(i)$ using *SJF* on a speed x_i processor, and imagine that all these executions take place in parallel. Notice that the total speed up required is $\sum_{i=1}^{\log_{1+\epsilon}(B/\epsilon)} x_i$ which is at most $1+\epsilon$. By, lemma 2.7, $S^p(\textit{SJF}(\mathcal{L}(i)), x_i)$ is at most $(1/x_i^p)S^p(\textit{SJF}(\mathcal{J}), 1)$. By simple algebraic calculation it can be seen that

$$\sum_i (\frac{1}{x_i})^p = O(\frac{1}{\epsilon^p}\log_{1+\epsilon}^{p+1} B)$$

Since the optimum schedule for $\mathcal{L}$ is no worse than the schedule constructed above the result follows. ∎

Combining equations 4.3, 4.5 and lemma 4.11, we get that

$$S^p(\textit{SETF}(\mathcal{I}), (1+\epsilon)^2) = O(\frac{(1+\epsilon)^{3p}}{\epsilon^p} \cdot \log_{1+\epsilon}^{p+1} B) S^p(\textit{SJF}(\mathcal{J}), 1) \tag{4.6}$$

Now by Theorem 2.3 we have that

$$S^p(\textit{SJF}(\mathcal{J}), 1+\epsilon) = O(1/\epsilon^p) S^p(\textit{Opt}(\mathcal{J}), 1) \tag{4.7}$$

Also, since each job $J_i \in \mathcal{J}$ has size at most $(1+\epsilon)^2$ times more than the corresponding job $J_i \in \mathcal{I}$ (and is not smaller), we trivially have that

$$S^p(\textit{Opt}(\mathcal{J}), (1+\epsilon)^2) \leq S^p(\textit{Opt}(\mathcal{I}), 1) \tag{4.8}$$

Now combining equations 4.6, 4.7 and 4.8 it follows that

$$S^p(\textit{SETF}, (1+\epsilon)^5, \mathcal{I}) = O(\frac{1}{\epsilon^{2p}} \cdot \log_{1+\epsilon}^{p+1} B) S^p(\textit{Opt}(\mathcal{I}), 1)$$

or equivalently that

$$S^p(\textit{SETF}, (1+\epsilon)^5, \mathcal{I}) = O(\frac{1}{\epsilon^{3p+1}} \cdot \log^{p+1} B) S^p(\textit{Opt}(\mathcal{I}), 1)$$

This proves Theorem 4.1.

4.6 Concluding Remarks

Our result that *SJF* is a $(1+\epsilon)$-speed, $O(1/\epsilon)$-competitive algorithm for minimizing the ℓ_p norms of stretch in the clairvoyant setting has been recently extended by Chekuri, Khanna and Kumar [22]. They show that load balancing scheme of [4] is a $(1+\epsilon)$-speed, $O(1/\epsilon)$-competitive algorithm for minimizing ℓ_p norms of stretch clairvoyantly on multiple machines.

The important open problems related to minimizing stretch are the following:

Open Problem 4.4 Does there exist a $(1+\epsilon)$-speed, $O(\log B)$-competitive algorithm for minimizing the total stretch non-clairvoyantly?

Open Problem 4.5 Does there exist an offline polynomial time exact algorithm for minimizing stretch on a single machine?

Minimizing the ℓ_p norms of stretch in the offline case is totally open.

Open Problem 4.6 Is there an approximation algorithm with non-trivial guarantees for minimizing the ℓ_p norms of stretch in the clairvoyant setting?

A polynomial time approximation scheme for minimizing total stretch on a single machine is known [18]. However nothing is known for the multiple machine case.

Open Problem 4.7 Is there a polynomial time approximation scheme for minimizing stretch on multiple machines?

Chapter 5

Minimizing Flow Time on Multiple Machines

5.1 Introduction

In this chapter we consider the offline problems of minimizing the total flow time and minimizing the maximum flow time on multiple machines.

While ***SRPT*** is the optimum algorithm for minimizing the total flow time on a single machine, the problem is $\mathcal{NP}$-Hard for the case of $m \geq 2$ machines. The first non-trivial result for the problem was an $O(\log(\min(n/m, B)))$ competitive algorithm for identical parallel machines due to Leonardi and Raz [55]. They also show a tight lower bound of $\Omega(\min(\log n/m, \log B))$ on the competitive ratio of any online algorithm. The algorithm of [55] has been extended to various other restricted settings (such as eliminating migrations [6] or immediately dispatching a job to a machine as it arrives [4]). Interestingly however, even in the offline case these algorithms remain the best known algorithms for the problem. Even for the case of $m = 2$, it is not known whether an $O(1)$ approximation algorithm for minimizing total flow time exists. On the other hand, only the $\mathcal{NP}$-Hardness of the problem is known. In particular, it is not even known if the problem is $\mathcal{APX}$-Hard. Obtaining an $O(1)$ approximation for this problem (even for the case of $m = 2$) has been a major open problem in scheduling [67, 55, 6].

Our main result is an algorithm for minimizing total flow time which produces a $(1 + \epsilon)$ approximate solution and has running time $n^{O(m \log n/\epsilon^2)}$. Thus for a fixed m, this gives a quasi-polynomial time approximation scheme. Our result suggests that a polynomial time approximation scheme is likely for minimizing total flow time on $O(1)$ machines.

The above result assumes that the machines are identical and that all jobs have equal weight. We extend it to the case when the machines are unrelated (a job could have different processing time on different machines) and additionally jobs are weighted. In this case, we give an approximation scheme with running time $n^{O(m \min(\log^2 nW, \log^2 nB)/\epsilon^3)}$. Prior to our work, Chekuri and Khanna gave an approximation scheme with a running time of $O(n^{\log B \log W/\epsilon^3})$ for a single machine. Thus our result extends the result of [21] to the case of multiple machines with almost matching running time.

The main idea used to obtain the above results is a technique which allows us to store the approximate state under ***SRPT*** for any subset of jobs using $O(\log^2 n)$ bits of information[1]. Moreover, this state descrip-

[1]Reducing this to $O(\log n)$ would imply a polynomial time approximation scheme.

tion has sufficient information to allow us to compute the new state as time progress and jobs are worked upon or when new jobs arrive. Thus, for each time t (we show later we only need to consider $O(n^3)$ values of time) and for each possible configuration, we can compute the best possible total flow time achievable using dynamic programming.

The running time in the above algorithms has an exponential dependence in m. We show that it is unlikely that this can be improved substantially (at least for the weighted case). In particular, we show that, if m is part of the input, then obtaining an $n^{o(1)}$ approximation algorithm with running time $n^{polylog(n,m)}$ for weighted flow time (even for the special case when W and B are polynomially bounded in n) would imply that $\mathcal{NP} \subseteq \mathcal{DTIME}(n^{polylogn})$.

Finally, we consider the problem of minimizing the maximum flow time on multiple machines. Again, while ***FIFO*** is optimal for the single machine case, the problem is $\mathcal{NP}$-Hard for $m \geq 2$. Our main result is a polynomial time approximation scheme for a constant number of unrelated machines. In particular, our algorithm produces a $(1+\epsilon)$ approximate solution and has running time $n^{O(m/\epsilon)}$. The only directly related work is for the problem of minimizing the maximum flow time on *identical* machines. For this problem Bender et al [17] give a $3 - 2/m$ competitive online algorithm, that in fact works for arbitrary m.

Observe that in the special when all the release times are 0, the problem of minimizing the maximum flow time reduces to the classic problem of minimizing the makespan on multiple unrelated machines (also referred to as the Generalized Assignment Problem in the literature). It is known that minimizing makespan on multiple unrelated machines is $\mathcal{APX}$-hard, when m is a part of the input [54]. This implies that dependence on m in the running time of our algorithm cannot be improved substantially.

Before we begin, we describe some extra terminology used in the multiple machine setting. In the unrelated machines setting, a job could possibly have different processing requirement on each machine. In this case, the processing time of a job J_j is described by an m-tuple $(p_{j1}, \dots, p_{jm})$, where p_{jk} is the processing time of J_j on machine k. An important issue while scheduling on multiple machines is that of migration. A schedule is called *migratory* if it can move partially executed jobs from one machine to another. Migration of jobs is usually considered unattractive as it incurs a huge overhead in practice. Throughout this chapter we only consider non-migratory algorithms.

5.2 Total Flow Time

In this section we consider the problem of minimizing the total flow time on multiple machines when all machines are identical and all jobs are unweighted.

Our high level idea is the following, we first show that the input instance can be rounded such that all job sizes are integers in the range $[1, n^2/\epsilon]$ and all the release times are integers in the range $[1, n^3/\epsilon]$. We then show how to store the approximate state under ***SRPT*** for any subset of jobs using $O(\log^2 n)$ bits of information. Finally, we show how this implies an approximation scheme for the problem.

Let $\mathcal{I}$ be a problem instance with largest job size B. Without loss of generality we can assume that all release dates are at most nB and that all jobs finish execution by time $2nB$ (otherwise we could reduce the problem into two disjoint problems). Let Opt denote the optimal schedule, we also abuse notation and use Opt to denote the total flow time under the optimal schedule.

Lemma 5.1 *Given $\mathcal{I}$, rounding up the job sizes and release dates to a multiple of $\epsilon B/n^2$ only increases the optimal cost of this rounded instance by a factor of $(1+2\epsilon)$.*

Proof. Given a schedule for the original instance, rounding up the job sizes adds at most $n \cdot \epsilon B/n^2 = \epsilon B/n$ to the flow time of each job. Similarly rounding up the release dates adds at most $\epsilon B/n^2$ to the flow time of each job. Thus, the total flow time is affected by at most $2\epsilon B \leq 2\epsilon Opt$. ■

Let $\mathcal{I}'$ denote this rounded instance. By Lemma 5.1 we can assume that all job sizes are integers in the range $[1, n^2/\epsilon]$. Similarly, as no release date in $\mathcal{I}$ is more than $2nB$, and the time is rounded to a multiple $\epsilon B/n^2$, all release dates in $\mathcal{I}'$ are integers in the range $[1, n^3\epsilon]$. We will obtain a schedule $S(\mathcal{I}')$ with optimum total flow time for $\mathcal{I}'$. Clearly, all events (arrivals and departures) under $S(\mathcal{I}')$ at integral times in the range $[1, 2n^3/\epsilon]$. Also, a schedule $S(\mathcal{I})$ for $\mathcal{I}$ follows from $S(\mathcal{I}')$ and that the total flow time under $S(\mathcal{I})$ is at most that under $S(\mathcal{I}')$. To see this, obtain $S(\mathcal{I})$ from $S(\mathcal{I}')$ by a scheduling a job in $\mathcal{I}$ only when the corresponding job in $\mathcal{I}'$ is scheduled. If the job in $\mathcal{I}$ finishes before the corresponding job in $\mathcal{I}'$ (since the job in $\mathcal{I}'$ could be larger), then the scheduler idles for that time. Finally, since the release times for a job in $\mathcal{I}'$ is no earlier than the corresponding job in $\mathcal{I}$, the schedule produced thus is a feasible schedule for $\mathcal{I}$. Henceforth, we only consider $\mathcal{I}'$.

We now describe how to compute an approximately optimal schedule (i.e. up to a factor of $1 + O(\epsilon)$) for $\mathcal{I}'$. We say that a job with size p_j lies in class i, iff $(1+\epsilon)^{i-1} \leq p_j < (1+\epsilon)^i$. This divides the jobs into $O(\frac{1}{\epsilon}\log n)$ classes. Note that the class of a job depends only on its initial processing time and hence does not change with time. Consider the optimum algorithm. Let $S_k(i,t)$ denote the jobs of class i on machine j which are alive at time t. The lemma below gives an important property of the optimal schedule.

Lemma 5.2 *For all $k = 1, \dots, m$ and at all times t, at most one job in $S_k(i,t)$ has a remaining processing time that is not in the range $(1+\epsilon)^{i-1}$ and $(1+\epsilon)^i$.*

Proof. As the shortest remaining processing time (SRPT) is optimal for minimizing the total flow time on a single machine, we can assume that on each machine, the jobs assigned to it are processed in the SRPT order.

Let us consider the class i jobs on a machine k. Since at any time we execute the job with least remaining time, if a job from this class is executed, clearly it must be the one which has the least remaining time in $S_k(i,t)$. Suppose for the sake of contradiction that there are two jobs with remaining time less than $(1+\epsilon)^{i-1}$, say J_1 and J_2. Consider the time t' when J_1 was worked upon and its remaining became less than $(1+\epsilon)^{i-1}$. If J_2 had remaining time less then $(1+\epsilon)^{i-1}$ at t', then J_1 should not have been worked upon (according to *SRPT*). If J_2 has not arrived by t' or its remaining time was greater than $(1+\epsilon)^{i-1}$, then the algorithm must have worked on J_2 during t' and the current time even though J_1 has remaining processing time less then J_2. ■

We now define the state of the algorithm $Q(t)$ at time t. For each class i and each machine j, we store $\frac{1}{\epsilon} + 1$ numbers: the first $\frac{1}{\epsilon}$ are the remaining processing times of the $\frac{1}{\epsilon}$ jobs in $S_j(i,t)$ with the largest remaining processing time; the last entry is the sum of the remaining processing times of the rest of the jobs in $S_j(i,t)$. Notice that as each job has size at most n^2/ϵ, there are at most $(n^2/\epsilon)^{1/\epsilon} = O(n^{2/\epsilon})$ possible choices for the first $1/\epsilon$ entries and n^3/ϵ possible choices for the sum of remaining sizes. Finally, since there there are at most $O(\log n/\epsilon)$ classes and m machines, the total number of distinct states at any step is at most $n^{O(m \log n/\epsilon^2)}$.

The following lemma shows how this information helps us in estimating the number of unfinished jobs at any point of time.

Lemma 5.3 *We can estimate the number of jobs in the system at time t to within a factor of $(1+2\epsilon)$ using the information in $Q(t)$.*

Proof. If there are fewer than $1/\epsilon$ jobs in level i, we know their number precisely because we store their remaining processing times precisely. Let us now consider the case when there are more than $1/\epsilon$ jobs in some level i.

Suppose at first that the remaining processing times of all these jobs lies between $(1+\epsilon)^{i-1}$ and $(1+\epsilon)^i$. Then, by assuming that all the jobs have size $(1+\epsilon)^{i-1}$ and computing the number of jobs using the total remaining processing time, our estimate is off from the correct number by at most a factor of $(1+\epsilon)$.

Finally, as at most one job in every level i lies outside the range $(1+\epsilon)^{i-1}$ and $(1+\epsilon)^i$. Thus our estimate above could be off by another job. However, there are at least $1/\epsilon$ unfinished jobs. Thus we get an estimate within a factor of $(1+2\epsilon)$. ∎

Thus the algorithm to compute the approximately optimal schedule will be a large dynamic program which will have a state for each time unit and each of the possible configurations of $Q(t)$ at that time. Thus there will be at most $O(n^3/\epsilon)$ times $n^{O(m \log n/\epsilon^2)}$ states. As usual, with each state we store the value of the least total flow time incurred to reach that state.

To complete the description of the dynamic program, we only need to show how to update the state of the algorithm with time. When a new job arrives, we have m choices for the machine to which this job can be assigned. Once a machine is decided, the size of the job determines the class to which it belongs. Also, it is straightforward to update the state, as either the job is added to the first $1/\epsilon$ jobs in $S_j(i,t)$, or else if it is smaller that the $1/\epsilon$ largest jobs in its class, then its size is added to the $(1/\epsilon+1)^{th}$ entry. Now consider the case when there are no arrivals. When the algorithm works on a job in level i on machine j, if there are $1/\epsilon$ or fewer jobs in level i, then we simply decrement the smallest of the remaining time entries by 1. Else, if there are more than $1/\epsilon$ jobs, we decrement the total remaining processing time entry by 1. In general, we will not know in which level the job with the least remaining processing time is present (as we do not have this information in our state description). However, we can try out all possible $O(\log n/\epsilon)$ choices for the different levels for each machine j. The we have $O(\log n/\epsilon)^m$ total possible choices at each time step.

Finally, have computed the approximately optimal value of flow time, it is straightforward to compute the path used to reach this value, which gives us the $(1+2\epsilon)$ optimal schedule for $\mathcal{I}'$. Finally by Lemma 5.1 this implies that

Theorem 5.4 *The above algorithm gives a $(1+\epsilon)$ approximation for the problem of minimizing total flow time on m identical machines and runs in time $n^{O(m \log n/\epsilon^2)}$.*

5.3 Extensions

We now give an algorithm for the more general case when the machines are unrelated and our measure is weighted flow time. Let p_j^* denote $\min_k p_{jk}$. Thus p_j^* denotes the minimum size of j among all the machines. Let $Q = \sum_j p_j^*$, and suppose that the weights are integers in the range $1, \ldots, W$. For unrelated machines, we define $B = \max_{i \neq j} p_i^*/p_j^*$. Note that if the machines are identical, this definition of B corresponds exactly to the maximum of minimum job size ratio. Also it is easily seen that $Q \leq nB \min_i p_i^*$.

We will give an algorithm that produces a $(1+\epsilon)$ approximate solution and has running time $n^{O(m \min(\log nB, \log nW)^2/\epsilon^3)}$. Our algorithm will consist of two different algorithms. First is an algorithm which has running time $n^{O(m \log^2 nW/\epsilon^3)}$ for the case when $W \leq B$. The other algorithm has running time $n^{O(m \log^2 nB/\epsilon^3)}$ for the case when $W \geq B$

We first consider the case when $W \leq B$. By assigning each job to the machine where it takes the least time, it is clear that nWQ is an upper bound on Opt. Similarly, Q is a lower bound on Opt. Hence we will assume that our algorithm simply never assigns a job J_j to machine k if $p_{jk} \geq 2nWQ$.

Next, as in lemma 5.1, it easily follows that rounding each p_{ij} and release date up to the next multiple of $\epsilon Q/(Wn^2)$ affects the total flow time of each job by at most $\epsilon Q/Wn$. Since the maximum weight of any job is at most W, this affects the total weighted flow time by at most $\epsilon Q \leq \epsilon Opt$. Moreover, we can also assume that the release date of any job is at most $2nQ$ (otherwise the problem can be decomposed into two disjoint instance). Thus after rounding the release dates and sizes to multiples of $\epsilon Q/(Wn^2)$, our algorithm needs to consider only $O(n^3 W/\epsilon)$ time steps.

Next we round up the weights to powers of $(1+\epsilon)$. Clearly, this affects the solution by at-most a $(1+\epsilon)$ factor. Finally, observe that in the optimal schedule for this rounded instance, if we consider a particular machine and restrict our attention to time intervals when jobs from a particular weight class are executed, then these jobs are executed in the SRPT order.

With these observations we can directly give an algorithm based on the ideas in Section 5.2. For each machine and each weight class, we maintain the states under SRPT. Since there are $O(\log W/\epsilon)$ weight classes, and there are $O(\log nW/\epsilon)$ size classes the number of states at each time step is bounded by $n^{O(m \log^2 nW/\epsilon^3)}$. When a job arrives, there are m choices corresponding to the machines it can be assigned. If there are no arrivals, for each machine, we need to decide which weight class to work on, and with each weight class which size class to work on. Thus there $(\log^2 nW/\epsilon^2)^m$ choices to choose from at each time step. Finally, since all the release dates and sizes are integers between 1 and $O(n^3 W/\epsilon)$, our algorithm can be implemented directly as a dynamic program of size Wn^3/ϵ times $n^{O(m \log^2 nW/\epsilon^3)}$. This gives the desired bound on the running time.

We now consider the case when $B \leq W$. Defining Q as previously we see that again Opt is at most nQW. Since $Q \leq nB \min_i p_i^*$, it follows that each job has size at least Q/nB on each machine. Finally, since the maximum weight is W, it follows that Opt is also lower bounded by WQ/nB.

Our algorithm now is as follows. We only consider jobs with weights between $\epsilon W/n^2 B$ and W. Jobs with weight below $\epsilon W/n^2 B$ will be added to the schedule arbitrarily. Since each job has flow time at most Q, this adds at most $\epsilon WQ/nB \leq \epsilon Opt$. Similarly, rounding the job sizes and release dates to multiples of $\epsilon Q/n^3 B$ affect the total weighted flow time by at most $n^2 W \cdot \epsilon Q/n^3 B = \epsilon WQ/nB \leq \epsilon Opt$.

Thus after rounding we have an instance where the job sizes and release dates are integers between 1 and $O(n^3 B)$ and the job weights are in the range $W/n^2 B$ and W. Thus there are $O(\log nB/\epsilon)$ weight classes and $O(\log nB/\epsilon)$ sizes classes. Applying the algorithm described above for the case $W \leq B$ and observing that all release dates are bounded by nB, we get that the running time of the algorithm is $n^{O(m \log nB^2/\epsilon^3)}$

Thus we have shown that

Theorem 5.5 *There is an algorithm for minimizing weighted flow time on multiple machines that produces a $(1+\epsilon)$ approximate solution and runs in time $n^{O(m \min(\log nB, \log nW)^2/\epsilon^3)}$.*

5.4 Dependence on the number of machines

It is known that both total unweighted flow time and maximum flow time on unrelated machines are $\mathcal{APX}$-Hard for arbitrary m [40, 54]. However, an approximation scheme with a polynomial dependence on m might be possible for identical parallel machines. We show a related but weaker negative result that, if m is a part of the input, minimizing total weighted flow time is $\mathcal{APX}$-hard, even when both B and W are polybounded in n.

Consider an instance of 3-Partition (SP15) in [31]. This consists of a set A of $3m$ elements, an integer bound $B > 0$; for each $x \in A$ a integer size $s(x)$ s.t. $B/4 < s(x) < B/2$ and s.t. $\sum_{x \in A} s(x) = mB$. The question is whether A can be partitioned into m disjoint sets $A_1, A_2, \ldots, A_m$ such that, for each $1 \leq i \leq m$, A_i has 3 elements and $\sum_{a \in A_i} s(a) = B$. 3-Partition is known to be strongly $\mathcal{NP}$-Complete. In particular, it is $\mathcal{NP}$-complete for $B = O(m^4)$.

Given an instance of 3-Partition, we transform it as follows. There are m machines. Each element $x \in A$ corresponds to a job, with size $s(x)$, weight m and is released at time $t = 0$. Next at each time instance $t = B + i/m^2$, for $i = 0, 1, 2, \ldots, Bm^5$, m jobs each with size $1/m^2$ and weight $1/m$ are released. Thus the total number of these small jobs is Bm^6.

If the instance of 3-partition has a solution, then we construct a schedule as follows: We schedule the jobs that arrive at $t = 0$ according to the solution to the instance of 3-partition. Since each job finishes by time $t = B$, we can schedule the jobs of weight $1/m$ as they arrive. Now, each of the weight m job has flow time at most $3B$, and each of the Bm^6 jobs of weight $1/m$ has flow time $1/m^2$. Thus the total weighted flow time is $3m \cdot 3B \cdot m + 1/m^3 \cdot Bm^6 = O(Bm^3)$.

On the other hand if there is no solution to the 3-partition instance, there is at least one weight m job (call it J) unfinished by time B. In particular, the job J has at least one unit of work unfinished by time B. Consider the situation by time $Bm^3/2$, if J is still there, it contributes at least $Bm^4/2$ to the flow time, else there are least m^2 jobs of size $1/m^2$ and weight $1/m$ piled up during the time interval $[Bm^3/2, Bm^3]$, thus contributing $O(Bm^4)$ to the total weighted flow time.

As $B = O(m^4)$ and and $W = m^2$ and $n = Bm^6 = O(m^{10})$, W and B are polybounded in the number of jobs, and we have an inapproximability factor of $\Omega(n^{1/10})$.

This implies that,

Theorem 5.6 *There does not exist an $n^{o(1)}$ approximation for minimizing weighted flow time on multiple machines if m is a part of the input, even if W and B are polynomially bounded in n.*

As a corollary, we also obtain that,

Theorem 5.7 *There cannot exist an algorithm for weighted flow time on multiple machines (with W and B polynomially bounded in n) that runs in time $2^{polylog(n,m)}$ and is an $n^{o(1)}$ approximation unless $\mathcal{NP} \subseteq \mathcal{DTIME}(n^{polylog(n)})$.*

This shows that for the problem of minimizing of minimizing the total weighted flow time on multiple machines (Theorem 5.5), the dependence of the running time on m in our algorithm is unlikely to be improved.

5.5 Maximum Flow Time

In this section we consider minimizing the maximum flow time on m unrelated machines. We give an algorithm that runs in time $O(n^{m/\epsilon})$ and produces a $(1+\epsilon)$ approximation. We begin with some easy observations.

1. As *FIFO* is optimum for minimizing the maximum flow time on a single machine. It follows that for every machine, the jobs assigned to that machine are executed in *FIFO* order.

2. As in Section 5.3, let p_j^* denote $\min_k p_{jk}$ and $Q = \sum_j p_j^*$. By assigning each job to the machine where it takes the least time, it is clear that Q is an upper bound on Opt. Furthermore, we can assume that all the release dates are at most Q and all jobs finish execution by time $2Q$ (else, the problem can be reduced to two smaller problems). Finally, since there are n jobs, there is some j such that $p_j^* \geq Q/n$ and hence this job has size at least Q/n on every machine. Thus Q/n is a lower bound on Opt.

3. Let $\delta > 0$ be an arbitrary real number. Note that rounding up the release date of each job to the next multiple of δ increases the maximum flow time by at most δ. Similarly, rounding up each p_{ij} to the next multiple of δ and increase the maximum flow time by at most $n\delta$.

We now describe the algorithm. If some $p_{jk} > 2Q$ for some j, k, we just set $p_{jk} = \infty$. Choose $\delta = \epsilon Q/4n^2$. Round each p_{jk} and r_j up to the next multiple of δ. Time increases in multiples of δ, and all events take place only at multiples of δ. Thus, we need to consider only $O(n^2)$ time steps.

The algorithm maintains a state of the $(val, t, w_1, w_2, \ldots, w_m)$ where t denotes the time, w_j represents the total work present on each machine j, $1 \leq j \leq m$ at time t, and val is minimum value of the maximum flow time that can be achieved at time t and for the particular values of w_i.

It is trivial to update the state of the algorithm, at each time step or event. If there is an arrival J_i at time t, and if the algorithm assigns it to machine j, then the algorithm updates the state as $w_j = w_j + p_{ij}$ and $val = \max\{val, w_j + p_{ij}\}$. If no arrival takes place, the algorithm simply decrements w_i for each non-zero w_i.

Thus the algorithm is a dynamic program of size $O(n^3/\epsilon)$ times $n^{O(m/\epsilon)}$ and updating an entry for a state requires $O(m)$ time. Thus we have that

Theorem 5.8 *The maximum flow time on multiple unrelated machines and arbitrary release dates can be approximated to within a factor of* $(1+\epsilon)$ *in time* $n^{O(m/\epsilon)}$.

5.6 Open Problems

The outstanding open problem is the following:

Open Problem 5.8 Is there a constant factor polynomial time approximation algorithm, or even better, a PTAS, for minimizing the total flow time on multiple identical machines?

The above problem is open even for $m = 2$.

Our $QPTAS$ for a constant number of machines suggests that a PTAS is likely for this problem. However, it could be the case that there is no polynomial time approximation scheme, or not even a constant factor approximation algorithm, when m is part of the input. Settling this would be interesting.

Open Problem 5.9 Is the problem of minimizing unweighted flow time on multiple machines $\mathcal{APX}$-Hard, if m is a part of the input?

For minimizing the maximum flow time, the obvious open question is

Open Problem 5.10 Does there exist a polynomial time approximation scheme for minimizing the maximum flow time on m identical machines when m is a part of the input?

Chapter 6

Scheduling for Flow-Time with Admission Control

6.1 Introduction

In this chapter, we consider the problem of scheduling on a single machine to minimize flow time where the jobs can be cancelled at some cost. Formally, for each job, the processor pays for each time unit the job is present in the system, and additionally a cancellation cost is charged if this job is ever cancelled. For instance, if a job arrives at time 0, and then at time 3 we decide to cancel it, we pay $c + 3$, where c is the job cancellation cost of this job. Equivalently, at each time step we pay for the number of unfinished jobs currently in the system plus any cancellations we decide to perform. One motivation for this problem arises from trying to formalize the scheduling decisions faced by a person in real life, where one also has the flexibility to accept or reject a request. Ideally, one would like not to have too many jobs on ones list of things to do (as this causes jobs to be delayed), or equivalently, one would like to have a small flow time, and rejecting some fraction of jobs might be necessary for this to happen (but of course one cannot ignore all work one is asked to do).

We also consider a more strict measure called *job-idle time*. In the job-idle time measure, we pay at each time step for the number of jobs currently in the system minus one (the one we are currently working on), or zero if there are no jobs in the system. Because job idle time is smaller than flow time, it is a strictly harder problem to approximate, and can even be zero if jobs are sufficiently well-spaced. Note that for the flow time measure, we can right away reject jobs that have size more than c, because if scheduled, these add at least c to the flow-time. However, this is not true for the job-idle time measure.

One issue with our definition of job-idle time is that it gives the machine credit for time spent on processing jobs that are later rejected. For example, if we get a job at time 0, work on it for 3 time units, and reject it at time 5, we pay $c + 2$ rather than $c + 5$. A natural alternative would be to define the cost so that no credit is given for time spent processing jobs that end up getting rejected. Unfortunately, that definition makes it impossible to achieve any finite competitive ratio. In particular, if a very large job arrives at time 0, we cannot reject it since it may be the only job and Opt would be 0; but, then if unit-size jobs appear at every time step starting at time tc, we have committed to cost tc whereas Opt could have rejected the big job at the start for a cost of only c. In any case, our definition of job-idle time is a strictly harder measure to approximate than flow time, and in some cases it produces cleaner results.

Our results are organized as follows: In section 6.2, we first consider a simpler problem where all the jobs have the same rejection costs. We call this the *Uniform Penalty Model*. For this case, we give a 2-competitive online algorithm for flow time and job-idle time with penalty. We also give a matching lower bound for job-idle time. In Section 6.3, we consider the setting where jobs have weights and we consider weighted flow time. We look at this problem in two models: In the first model, the cancellation costs are identical no matter what the weight of the jobs. In the second model, the rejection costs are proportional to the job's weight. For both these cases we give an $O(\log^2 W)$ competitive algorithm. In Section 6.4, we consider the case when jobs could have arbitrary penalties. We show that there cannot be any randomized algorithm which is $n^{o(1)}$ competitive in this setting. We thus consider the problem in the resource augmentation model. In fact, we consider a more general problem where we allows jobs to have arbitrary weights and arbitrary rejection penalties. For this case we give a $(1+\epsilon)$-speed, $O(\frac{1}{\epsilon}(\log W + \log C)^2)$-competitive algorithm, where C is the ratio between the maximum to minimum rejection penalty.

Related Previous Work

Admission control has been studied for a long time in circuit routing (see, e.g., [19]). In these problems, the focus is typically on approximately maximizing the throughput of the network. In scheduling problems, the model of rejection with penalty was first introduced by Bartal et al [12]. They considered the problem of minimizing makespan on multiple machines with rejection and gave a $1+\phi$ approximation for the problem where ϕ is the golden ratio. Variants of this problem have been subsequently studied by [68, 39]. Seiden [68] extends the problem to a pre-emptive model and improves the ratio obtained by [12] to 2.38.

More closely related to our work is the model considered by Engels et al [29]. They consider the problem of minimizing weighted *completion* time with rejections. However, there are some significant differences between their work and ours. First, their metric is different. Second, they only consider the offline problem and give a constant factor approximation for a special case of the problem using LP techniques.

Preliminaries

To get a feel for this problem, notice that we can model the classic ski-rental problem as follows. Two unit-size jobs arrive at time 0. Then, at each time step, another unit-size job arrives. If the process continues for less than c time units, the optimal solution is not to reject any job. However, if it continues for c or more time units, then it would be optimal to reject one of the two jobs at the start. In fact, this example immediately gives a factor 2 lower bound for deterministic algorithms for job-idle time, and a factor $3/2$ lower bound for flow time.

To get a further feel for the problem, consider the following online algorithm that one might expect to be constant-competitive, but in fact does not work: Schedule jobs using *SRPT*, but whenever a job has been in the system for more than c time units, reject this job, incurring an additional c cost. Now consider the behavior of this algorithm on the following input: m unit size jobs arrive at time 0, where $m < c$, and subsequently one unit size job arrives in every time step for n steps. *SRPT* (breaking ties in favor of jobs arriving earlier) will schedule every job within m time units of its arrival. Thus, the proposed algorithm does not reject any job, incurring a cost of mn, while *Opt* rejects $m-1$ jobs in the beginning, incurring a cost of only $n+(m-1)c$. This gives a competitive ratio of m as $n \to \infty$.

We now state some properties of the optimal solution (*Opt*) which will be useful in deriving our results.

Fact 6.1 *If Opt rejects a job j, it is rejected the moment it arrives.*

Fact 6.2 *Given the set of jobs that Opt rejects, the remaining jobs must be serviced in the SRPT order.*

Fact 6.3 *In the uniform penalty model, if a job j is rejected, then it must be the job that currently has the largest remaining time.*

6.2 An Online Algorithm

In this section, we give online algorithms for minimizing flow time and job idle time in the uniform penalty model.

6.2.1 Minimizing Flow Time

Flow time of a schedule can be expressed as the sum over all time steps of the number of jobs in the system at that time step. Let ϕ be a counter that counts the flow time accumulated until the current time step. The following algorithm achieves 2-competitiveness for flow time with rejections:

The Online Algorithm. Starting with $\phi = 0$, at every time step, increment ϕ by the number of active jobs in the system at that time step. Whenever ϕ crosses a multiple of c, reject the job with the largest remaining time. Schedule active jobs in *SRPT* order.

Let the schedule produced by the above algorithm be S and the set of rejected jobs be R.

Lemma 6.4 *The cost of the algorithm is $\leq 2\phi$.*

Proof. This follows from the behavior of the algorithm. In particular, $F(S)$ is equal to the final value in the counter ϕ, and the total rejection cost $c|R|$ is also at most ϕ because $|R|$ increases by one (a job is rejected) every time ϕ gets incremented by c. ∎

The above lemma implies that to get a 2-approximation, we only need to show that $\phi \leq Opt$. Let us use another counter ψ to account for the cost of Opt. We will show that the cost of Opt is at least ψ and at every point of time $\psi \geq \phi$. This will prove the result.

The counter ψ works as follows: If Opt rejects a job, ψ gets incremented by c. Else, if $\phi = \psi$, then ϕ and ψ increase at the same rate (i.e. ψ stays equal to ϕ). Else ψ stays constant. By design, we have the following:

Fact 6.5 *At all points of time, $\psi \geq \phi$.*

Let $k = \lfloor \frac{\psi}{c} \rfloor - \lfloor \frac{\phi}{c} \rfloor$. Let n_o and n_a denote the number of active jobs in Opt and A respectively. Arrange and index the jobs in Opt and A in the order of decreasing remaining time. Let us call the k longest jobs of A *marked*. We will now prove the following:

Lemma 6.6 *At all times $n_o \geq n_a - k$.*

Lemma 6.6 will imply $Opt \geq \psi$ (and thus, 2-competitiveness) by the following argument: Whenever ψ increases by c, Opt spends the same cost in rejecting a job. When ψ increases at the same rate as ϕ, we have that $\psi = \phi$. In this case $k = 0$ and thus Opt has at least as many jobs in system as the online algorithm. Since the increase in ϕ (and thus ψ) accounts for the flow time accrued by the online algorithm, this is less than the flow time accrued by Opt. Thus the cost of Opt is bounded below by ψ and we are done.

We will prove Lemma 6.6 by induction over time. For this we will need to establish a suffix lemma. We will ignore the marked jobs while forming suffixes.

Let $P_o(i)$ (called a suffix) denote the sum of remaining times of jobs $i, \ldots, n_o$ in Opt. Let $P_a(i)$ denote the sum of remaining times of jobs $i+k, \ldots, n_a$ in A ($i, \ldots, n_a - k$ among the unmarked jobs). For instance, Figure 6.1 below shows the suffices for $i = 2$ and $k = 2$.

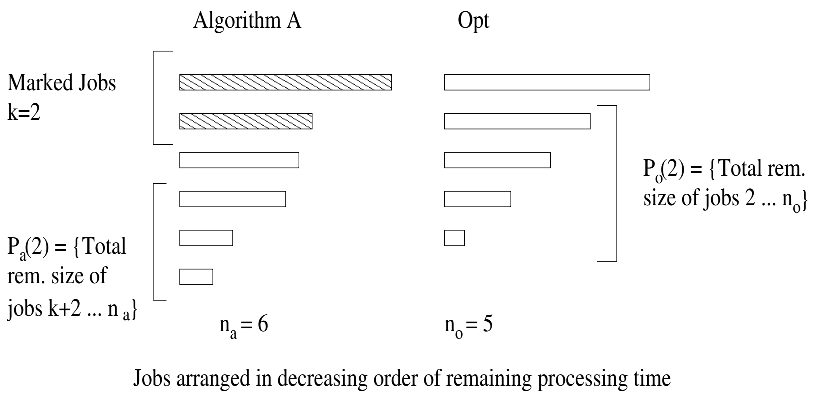


Figure 6.1: *Notation used in proof of Theorem 6.9*

Lemma 6.7 *At all times, for all i, $P_a(i) \leq P_o(i)$.*

Proof. (of Lemma 6.6 using Lemma 6.7) Using $i = n_a - k$, we have $P_o(n_a - k) \geq P_a(n_a - k) > 0$. Therefore, $n_o \geq n_a - k$. ■

Proof. (Lemma 6.7) We prove the statement by induction over the various events in the system. Suppose the result holds at some time t. First consider the simpler case of no arrivals. Furthermore, assume that the value of k does not change from time t to $t+1$. Then, as A always works on the job n_a, $P_a(i)$ decreases by 1 for each $i \leq n_a - k$ and by 0 for $i > n_a - k$. Since $P_o(i)$ decreases by at most 1, the result holds for this case.

If the value of k changes between t and $t+1$, then since there are no arrivals (by assumption), it must be the case that A rejects some job(s) and k decreases. However, note that rejection of jobs by A does not affect any suffix under A (due to the way $P_a(i)$ is defined). Thus the argument in the previous paragraph applies to this case.

We now consider the arrival of a job J at time t. If J is rejected by Opt, the suffixes of Opt remain unchanged and the value of k increases by 1. If J gets marked under A, none of the suffixes under A change either, and hence the invariant remains true. If J does not get marked, some other job with a higher remaining time than J must get marked. Thus the suffixes of A can only decrease.

If J is not rejected by Opt, we argue as follows: Consider the situation just before the arrival of J. Let C be the set of unmarked jobs under A and D the set of all jobs under Opt. On arrival of J, clearly J gets added to D. If J is unmarked under A it gets added to C else if it gets marked then a previously marked job $J' \in A$, with a smaller remaining time than J gets added to C. In either case, the result follows from Lemma 6.8 (see Proposition A.7, Page 120 in [56] or Page 63 in [36]), which is a result about suffixes of sorted sequences, by setting $\mathcal{C} = C$, $\mathcal{D} = D$, $d' = J$ and $c' = J$ or J'. ■

Lemma 6.8 *Let $\mathcal{C} = \{c_1 \geq c_2 \geq \ldots\}$ and $\mathcal{D} = \{d_1 \geq d_2 \geq \ldots\}$ be sorted sequences of non-negative numbers. We say that $\mathcal{C} \prec \mathcal{D}$ if $\sum_{j \geq i} c_j \leq \sum_{j \geq i} d_i$ for all $i = 1, 2, \ldots$. Let $\mathcal{C} \cup \{c'\}$ be* *the sorted sequence obtained inserting c' in $\mathcal{C}$. Then, $\mathcal{C} \prec \mathcal{D}$ and $c' \leq d' \Rightarrow \mathcal{C} \cup \{c'\} \prec \mathcal{D} \cup \{d'\}$.*

Thus we have the following theorem:

Theorem 6.9 *The above online algorithm is 2-competitive with respect to Opt for the problem of minimizing flow time with rejections.*

6.2.2 Minimizing Job Idle Time

Firstly note that the job idle time of a schedule can by computed by adding the contribution of the jobs waiting in the queue (that is, every job except the one that is being worked upon, contributes 1) at every time step.

The same online algorithm as in the previous case works for minimizing job idle time with the small modification that the counter ϕ now increments by the number of waiting jobs at every time step. The analysis is similar and gives us the following theorem:

Theorem 6.10 *The above online algorithm is 2-competitive with respect to Opt for the problem of minimizing job idle time with rejections.*

6.2.3 Varying Server Speeds

For a researcher managing his/her to-do list, one typically has different amounts of time available on different days. We can model this as a processor whose speed changes over time in some unpredictable fashion (i.e., the online algorithm does not know what future speeds will be in advance). This type of scenario can easily fool some online algorithms: e.g., if the algorithm immediately rejected any job of size $\geq c$ according to the current speed, then this would produce an unbounded competitive ratio if the processor immediately sped up by a large factor.

However, our algorithm gives a 2-approximation for this case as well. The only effect of varying processor speed on the problem is to change sizes of jobs as time progresses. Let us look at the problem from a different angle: the job sizes stay the same, but time moves at a faster or slower pace. The only effect this has on our algorithm is to change the time points at which we update the counters ϕ and ψ. However, notice that our algorithm is locally optimal: at all points of time the counter ψ is at most the cost of Opt, and $\phi \leq \psi$, irrespective of whether the counters are updated more or less often. Thus the same result holds.

6.2.4 Lower Bounds

We now give a matching lower bound of 2 for waiting time and 1.5 for flow time, on the competitive ratio of any deterministic online algorithm.

Consider the following example: Two jobs of size 1 arrive at $t = 0$. The adversary gives a stream of unit size jobs starting at $t = 1$ until the algorithm rejects a job.

Let x be the time when the algorithm first rejects a job. In the waiting time model, the cost of the algorithm is $x + c$. The cost of the optimum is $\min(c, x)$, since it can either reject a job in the beginning, or not reject at all. Thus we have a competitive ratio of 2.

The same example gives a bound of 1.5 for flow time. Note that the cost of the online algorithm is $2x + c$, while that of the optimum is $\min(x + c, 2x)$.

Theorem 6.11 *No online algorithm can achieve a competitive ratio of less than 2 for minimizing waiting time with rejections or a competitive ratio of less than 1.5 for minimizing flow time with rejections.*

6.3 Weighted flow time with weighted penalties

In this section we consider the minimization of weighted flow time with admission control. We assume that each job has a weight associated with it. Without loss of generality, we can assume that the weights are powers of 2. This is because rounding up the weights to the nearest power of 2 increases the competitive ratio by at most a factor of 2. Let $a_1, a_2, \ldots, a_k$ denote the different possible weights, corresponding to weight classes $1, 2, \ldots, k$. Let W be the ratio of maximum to minimum weight. Then, by our assumption, k is at most $\log W$. We will consider the following two models for penalty. The general case of arbitrary penalties is considered in the next section.

Uniform penalty: Jobs in each weight class have the same penalty c of rejection.

Proportional penalty: Jobs in weight class j have rejection penalty $a_j c$.

For both these cases, we give an $O(\log^2 W)$ competitive algorithm. This algorithm is based on the Balanced *SRPT* algorithm described in Chapter 3, Section 3.4 . We modify their algorithm to incorporate admission control. The modified algorithm is described below.

Algorithm Description: As jobs arrive online, they are classified according to their weight class. Consider the weight class that has the minimum total remaining time of jobs. Ties are resolved in favor of higher weight classes. At each time step, we pick the job in this weight class with smallest remaining time and schedule it.

Let ϕ be a counter that counts the total weighted flow time accumulated until current time step. For each weight class j, whenever ϕ crosses the penalty c (resp. $a_j c$), we reject a job with the largest remaining time from this class.

Analysis: We will imitate the analysis of the weighted flow time algorithm. First we give an upper bound on the cost incurred by the algorithm. Let $F(S)$ be the final value of counter ϕ. The cost of rejection, $c|R|$, is bounded by $k\phi$, because rejections $|R_j|$ in weight class j increase by 1 every time ϕ increases by c_j. Thus we have,

Lemma 6.12 *The total cost of the algorithm is* $\leq (k+1)\phi$

In order to lower bound the cost of optimal offline algorithm, we use a counter ψ. The counter ψ works as follows: Whenever *Opt* rejects a job of weight class j, ψ gets incremented by c_j. At other times, if $\phi = \psi$, then ϕ and ψ increase at the same rate (i.e. ψ stays equal to ϕ), otherwise, ψ stays constant. By design, we have the following:

Fact 6.13 *At all points of time,* $\psi \geq \phi$.

Now we show that ψ is a lower bound on $k \cdot Opt$. Let $m_j = \lfloor \frac{\psi}{kc_j} \rfloor - \lfloor \frac{\phi}{kc_j} \rfloor$. In both *Opt* and our algorithm, arrange active jobs in each weight class in decreasing order of remaining processing time. We call the first m_j jobs of weight class j in our algorithm as marked. Now ignoring the marked jobs, we can use Theorem 3.5 from Chapter 3. We get the following:

Lemma 6.14 *The total weight of unmarked jobs in our algorithm is no more than k times the total weight of jobs in Opt.*

Proof. The proof is essential similar to that of Lemma 3.4. However, due to rejections, we need to check a few more cases.

We first restate that lemma in terms suitable for our purpose. Let $B(j,l)$ and $P(j,l)$ denote a *prefix* of the jobs in our algorithm and *Opt* algorithm respectively. Then, we define the suffixes $\overline{B}(j,l) = J_a - B(j,l)$ and $\overline{P}(j,l) = J_o - P(j,l)$, where J_a and J_o are the current sets of jobs in our algorithm and the *Opt* algorithm respectively.

Lemma 6.15 **([7])** *The total remaining time of the jobs in the suffix $\overline{B}(j,l)$ is smaller than the total remaining time of the jobs in $\overline{P}(j,l)$.*

We now consider the cases that are not handled by Bansal *et al.*'s proof. If a job of weight class j arrives and *Opt* rejects it, then the set of jobs with *Opt* does not change. On the other hand, m_j increases by at least 1. In our algorithm, if the new job is among top m_j jobs in its weight class, then it is marked and set of unmarked jobs remains the same. If the new job does not get marked, the suffixes of our algorithm can only decrease, since some other job with higher remaining time must get marked.

Similarly, when our algorithm rejects a job of class j, then the number of marked jobs m_j reduces by 1. However, the rejected job had highest remaining time in the class j. Hence none of the suffixes change.

Thus, we have established that the suffixes in our algorithm are smaller than the corresponding suffixes in the *Opt* algorithm at all times. The argument from Theorem 3.5 gives us the result that weight of unmarked jobs in our algorithm is at most $k \cdot Opt$. ■

To finish the argument, note that when the *Opt* algorithm rejects a job of weight class j, *Opt* increases by c_j. And ψ increases by kc_j. On the other hand, when ψ and ϕ increase together, we have $\psi = \phi$. There are no marked jobs, since $m_j = 0$ for all j. The increase in ψ per time step is same as the weight of all jobs in our algorithm. As we saw in the Lemma 6.14, this is at most k times the total weight of jobs in *Opt* algorithm. Thus, the total increase in ψ is bounded by $k \cdot Opt$.

In conjunction with Lemma 6.12, this gives us that

Theorem 6.16 *For both the "Proportional Penalty Model" and the "Uniform Penalty Model", the algorithm described in Section 6.3 is an $O(\log^2 W)$ competitive algorithm for minimizing the weighted flow time with rejection costs.*

6.4 Weighted Flow time with Arbitrary Penalties

In this section we will consider the case when different jobs have different weights and different penalties of rejection. First we will show that even for the simpler case of minimizing unweighted flow time with two different penalties, no algorithm can obtain a competitive ratio of $o(n^{\frac{1}{4}})$ or $o(C^{\frac{1}{2}})$. A similar bound holds even if there are two different penalties and the arrival times of high penalty jobs are known in advance. Then we will give an online algorithm that achieves a competitive ratio of $O(\frac{1}{\epsilon}(\log W + \log C)^2)$ using a processor of speed $(1+\epsilon)$.

6.4.1 Lower Bounds

Theorem 6.17 *For the problem of minimizing flow time or job idle time with rejection, and arbitrary penalties, no (randomized) online algorithm can achieve a competitive ratio of $o(n^{\frac{1}{4}})$ or $o(C^{\frac{1}{2}})$. Even when there are only two different penalties and the algorithm has complete knowledge of the high penalty jobs (i.e., it knows the release times and sizes of these jobs in advance), no online (randomized) algorithm can achieve a competitive ratio of $o(n^{\frac{1}{5}})$.*

Proof. Consider the following scenario for a deterministic algorithm. The adversary gives two streams, each beginning at time $t = 0$. Stream1 consists of k^2 jobs, each of size 1 and penalty k^2. Stream2 consists of k jobs each of size k and penalty infinite.

Depending on what the online algorithm does by time k^2, the adversary will decide whether it should give a third stream of jobs. Stream3 consists of $m = k^4$ jobs, each of size 1 and penalty infinite.

Let y denote the total remaining work of jobs of Stream2 that are left at time $t = k^2$. The adversary gives Stream3 if $y \geq k^2/2$.

First consider the case when $y \geq k^2/2$. Observe that at time $t = k^2$, there are at least $k/2$ jobs remaining. At this time, the online algorithm will work on jobs of Stream3 (using ***SRPT***, since none of these will be rejected). Thus we get a cost of at least $mk/4 = k^5/4$. On the other hand, the optimal offline algorithm rejects all the jobs of Stream1 and finishes jobs of Stream2 during the first k^2 time steps. After time $t = k^2$ it can work on Stream3. Thus the total cost incurred by Opt is $k \cdot k + k^2 \cdot k^2 + m \cdot 1 = O(k^4)$. Thus, in this case the ratio of the optimal cost to the online cost is at $\Omega(k)$.

Second, if $y < k^2/2$, the adversary does not give Stream 3. Note that if $y < k^2/2$, this means that algorithm has spent more than $k^2/2$ time on Stream2, so less than $k^2/2$ jobs of Stream 1 could have been finished by $t = k^2$. Now suppose that the online algorithm rejects more than $k^2/4$ jobs. Then it incurs a penalty of at least $k^4/4$. On the other hand, if it rejects fewer than $k^2/4$ jobs, then at least $k^2/4$ jobs remain at time k^2, and finishing these adds up at least $\frac{1}{2}(\frac{k^2}{4})^2$ to the flow time. So the online algorithm incurs at least a cost of $\Omega(k^4)$. In this case, the offline algorithm can first finish Stream1 and then work on Stream2. Note that flow time of each Stream2 job is $k^2 + k$. Thus the total cost here is $k^2 \cdot 1 + k \cdot (k^2 + k) = O(k^3)$, which again implies a ratio of $\Omega(k)$ as compared with the optimal offline.

To obtain a lower bound on competitive ratio in terms of C, we simply replace the infinite penalty by a penalty of k^4 (thus, both jobs is Stream2 and Stream3 have a penalty of k^4). Note that if the online algorithm rejects $\Omega(k)$ jobs of Stream2 (penalty k^4), then it incurs a cost of at least $\Omega(k^5)$ and we get a a lower bound of k. If it does not reject $\Omega(k)$ jobs of Stream2, the argument above works exactly like before. Now since $C = k^4/k^2 = k^2$, we get a lower bound of $\Omega(C^{1/2})$. Clearly, the lower bound extends to the randomized case, as the adversary can simply send Stream3 with probability 1/2.

Now consider the case of known high penalty jobs. We consider an instance with two types of penalties: k^2 and infinity. The sizes and arrival times of the infinite penalty jobs are known in advance. Our example uses $O(k^5)$ jobs, and gives a lower bound of $\Omega(k) = \Omega(n^{1/5})$ on the competitive ratio.

Let Stream1 consist of k^2 jobs of size 1 and penalty k^2, and Stream2 consist of k jobs of size k and infinite penalty, as before. Both arrive at time $t = 0$. We redefine Stream3 to consist of k^5 jobs, each of size $1/k$ and penalty k^2. Note that the only infinite penalty jobs are in Stream 2, which by assumption are known to the algorithm in advance. Jobs in both Stream1 and Stream3, have the same penalty.

As before, it is easy to see that if Stream3 is not given, the optimum cost is $O(k^3)$. If the stream is given, it is optimal to reject all jobs in Stream1 and work only on Stream2 and Stream3, which gives a cost of $O(k^4)$.

In the online scenario, the adversary gives Stream3 at time $t = k^2$, if and only if the total remaining work of Stream2 jobs at that time is more than $k^2/2$. As before, if the remaining work of Stream2 is less than $k^2/2$, then the algorithm has either rejected $\Omega(k^2)$ jobs of Stream1, or delays them by $\Omega(k^2)$ time. In either case, the cost of the algorithm is at least $\Omega(k^4)$, implying a competitive ratio is at least $\Omega(k)$. If the remaining work of Stream2 is greater than $k^2/2$ and Stream3 is given, then either $\Omega(k^3)$ jobs of Stream3 need to be rejected, giving a penalty of $\Omega(k^5)$, or else, $\Omega(k)$ jobs of Stream2 are delayed for the duration of Stream3 i.e. $\Omega(k^4)$, which gives a cost of $\Omega(k^5)$. Thus, the ratio is again at least $\Omega(k)$. ■

6.4.2 Algorithm with Resource Augmentation

Now we will give a resource augmentation result for the weighted case with arbitrary penalties. Consider first, a fractional model where we can reject a fraction of a job. Rejecting a fraction f of job j has a penalty of fc_j. The contribution to the flow time is also fractional: If an f fraction of a job is remaining at time t, it contributes fw_j to the weighted flow time at that moment.

Given an instance of the original problem, create a new instance as follows: Replace a job j of size p_j, weight w_j and penalty c_j, with c_j jobs, each of weight w_j/c_j, size p_j/c_j and penalty 1.

Using the $O(\log^2 W)$ competitive algorithm for the case of arbitrary weights and uniform penalty, we can solve this fractional version of the original instance to within $O((\log W + \log C)^2)$. Now we use a $(1+\epsilon)$ speed processor to convert the fractional schedule back to a schedule for the original metric without too much blowup in cost, as described below.

Denote the fractional schedule output in the first step by S_F. The algorithm works as follows: If S_F rejects more than an $\epsilon/2$ fraction of some job, reject the job completely. Else, whenever S_F works on a job, work on the same job with a $(1+\epsilon)$ speed processor. Notice that when the faster processor finishes the job, S_F still has $1 - \epsilon/2 - 1/(1+\epsilon) = O(\epsilon)$ fraction of the job present.

We lose at most $2/\epsilon$ times more than S_F in rejection penalties, and at most $O(1/\epsilon)$ in accounting for flow time. Thus we have the following theorem:

Theorem 6.18 *The above algorithm is $O(\frac{1}{\epsilon}(\log W + \log C)^2)$-competitive for the problem of minimizing weighted flow time with arbitrary penalties on a $(1+\epsilon)$-speed processor.*

6.5 Conclusion

In this chapter, we give online algorithms for the problems of minimizing flow time and job idle time when rejections are allowed at some penalty, and examine a number of problem variants. There are a couple of problems left open by our work.

Open Problem 6.11 It would be interesting to close the gap between the 1.5 lower bound and our 2-competitive algorithm for minimizing flow time with uniform penalties.

Open Problem 6.12 Is the offline version of the problem $\mathcal{NP}$-Hard for arbitrary rejection penalties?

Chapter 7

Results for Completion Time Scheduling via Flow Time

7.1 Introduction

In this chapter we consider online algorithms for minimizing objective functions which depend on the completion times of jobs. While a performance guarantee on a measure based on completion time is not as useful as a guarantee on flow time, this line of research has been of significant academic interest and often leads to interesting insights and algorithmic techniques. A lot of these results can be found in the nice surveys on the topic [47, 69, 52, 33, 62, 64]. More recent results which also simplify and unify the previous work can be found in [1, 32, 24, 66, 57].

The completion time measure is usually quite robust with respect to various perturbations and changes in the input instance such as changes in the release dates or the processing times of jobs. This robustness has been exploited to give very general techniques which allow us to convert a result from a restricted setting to a more general setting. Two previous results which fall in this paradigm are the following:

- Shmoys, Wein and Williamson [70] consider the problem of scheduling jobs on parallel machines to minimize the maximum completion time in the online non-clairvoyant setting. They give a technique for converting an existing offline $\rho-$approximation for a problem into a $2\rho-$competitive online and non-clairvoyant algorithm.

- Hall, Symoys and Wein [34] consider the problem of minimizing the weighted sum of completion times in the online setting. They give a technique that converts a $\rho-$approximation algorithm for a related problem called the *Maximum Scheduled Weight Problem* (see [34] for the problem definition) into a $4\rho-$ approximation for the online completion time problem.

In this chapter, we give a result of similar flavor. We will give a technique to transform an algorithm for a flow time problem which possibly uses resource augmentation to obtain an algorithm for the corresponding completion time problem which does not use resource augmentation. Our transformation carries online algorithms to online algorithms and also preserves non-clairvoyance. As a corollary of this result we will obtain $O(1)$-competitive online clairvoyant and non-clairvoyant algorithms for minimizing the weighted ℓ_p norms of completion time. [1]

[1] The results in the chapter are from [10].

7.2 Preliminaries

We first make precise the notion of a completion time measure corresponding to a flow time measure. Given a schedule S for n jobs, this determines the flow times $f_1, \ldots, f_n$ and the completion times $c_1, \ldots, c_n$. Let $\mathcal{G}$ be some function that takes as input n real numbers and outputs another real number. Given a schedule S, we define the functions $\mathcal{F}$ and $\mathcal{C}$ as follows:

$$\mathcal{F}(S) = \mathcal{G}(f_1, f_2, \ldots, f_n)$$

$$\mathcal{C}(S) = \mathcal{G}(c_1, c_2, \ldots, c_n)$$

For example, if $\mathcal{G}(x_1, \ldots, x_n) = (\sum_i w_i x_i^p)^{1/p}$, then $\mathcal{F}$ and $\mathcal{C}$ are simply the weighted ℓ_p norms of flow time and completion time respectively.

Our technique for converting a flow time result to a completion time result will require two properties from the function $\mathcal{G}$.

Definition 7.1 *A function is said to be* scalable *if for any positive real number k, $\mathcal{G}(kx_1, \ldots, kx_n) = k\mathcal{G}(x_1, \ldots, x_n)$. In particular, if we scale all the flow times in a schedule by k times then $\mathcal{F}(S)$ increases by k times.*

We now motivate the next property that we require from the function $\mathcal{G}$. We first point out a somewhat surprising property of the ℓ_p norms of the completion time measure. While it is easy to see that minimizing the total weighted flow time (i.e. ℓ_p norm with $p = 1$) is equivalent to minimizing the total weighted completion time, this is not the case for $p > 1$. In particular, it could be the case that a schedule which is optimum for the $\sum_i f_i^2$ measure is sub-optimal for $\sum_i c_i^2$ measure and vice versa.

Consider the following instance with just two jobs. The first job has size 10 and arrives at $t = 0$, the second job has size 1 and arrives at $t = 8$. A simple calculation shows that in order to minimize the total flow time squared, it is better to first finish the longer job and then the smaller job. This incurs a total flow time squared of $10^2 + 3^2 = 109$, where as the other possibility which is to finish the small job as soon as it arrives an then finish the big job incurs a total flow time squared of $11^2 + 1^2 = 122$. On the other hand, if we consider completion time squared, finishing the larger job first incurs a cost of $10^2 + 11^2$. If instead if finish the smaller job first, this incurs a cost of $9^2 + 11^2$. Thus the optimal schedule for ℓ_p norms of flow time need not be optimal for ℓ_p norms of completion time and vice versa.

Definition 7.2 *We say that a function $\mathcal{G}$ is ρ-good if is satisfies the following condition:*

Given a problem instance $\mathcal{I}$ and any two arbitrary schedules S and S' for $\mathcal{I}$. If $\mathcal{F}(S) \leq c\mathcal{F}(S')$, then $\mathcal{C}(S) \leq \rho c\mathcal{C}(S')$.

7.3 Results

Our main result is the following:

Theorem 7.3 *Let $\mathcal{G}$ be a ρ-good function. If there is an s-speed, c-competitive online algorithm with respect to the measure $\mathcal{F}$ (derived from $\mathcal{G}$), then this algorithm can be transformed into another online algorithm which is 1-speed, ρcs-competitive with respect to the corresponding completion time measure $\mathcal{C}$. Moreover, non-clairvoyant algorithms are transformed into non-clairvoyant algorithms.*

We now describe the transformation:

Let A be a s-speed, c-competitive algorithm for a flow time problem. Let $\mathcal{I}$ be the original instance where job J_i has release date r_i and size p_i. The online algorithm (which we call B) is the defined as follows:

1. When a job arrives at time r_i, pretend that it has not arrived till time sr_i.
2. At any time t, run A on the jobs for which $t \geq sr_i$

Proof. (of Theorem 7.3) Let I' be the instance obtained from $\mathcal{I}$ by replacing job $J_i \in \mathcal{I}$ by a job J_i' that has release date sr_i and size sp_i. Also, let $\mathcal{I}''$ be the instance from $\mathcal{I}$ by replacing the job $J_i \in \mathcal{I}$ with a job J_i'' that has release date sr_i and size p_i.

Let $Opt(\mathcal{F}, \mathcal{I}, x)$ (resp $Opt(\mathcal{C}, \mathcal{I}, x)$) denote the flow time cost (resp completion time cost) of the optimum schedule on $\mathcal{I}$ run using an x speed processor. We first relate the values of the optimum schedules for $\mathcal{I}$ and $\mathcal{I}'$.

Fact 7.4 $Opt(\mathcal{C}, \mathcal{I}', 1) = sOpt(\mathcal{C}, \mathcal{I}, 1)$

Proof. (of Fact 7.4:) Since the release times and sizes in $\mathcal{I}'$ are scaled by s times that of $\mathcal{I}$, given a schedule S for $\mathcal{I}$, we can construct a schedule S' for $\mathcal{I}'$ such that if some event happens at time t in $\mathcal{I}$, then the corresponding event in $\mathcal{I}'$ happens at time st. By the scalability property of the function $\mathcal{G}$, the value of the schedule S' is exactly s times more than that for S. Similarly, given any schedule for $\mathcal{I}'$ we can construct a schedule for $\mathcal{I}$ with value exactly s times smaller. Thus the result follows. ■

By our resource augmentation guarantee for the algorithm A, we know that

$$\mathcal{F}(A, \mathcal{I}', s) \leq cOpt(\mathcal{F}, \mathcal{I}', 1)$$

By the ρ-goodness of $\mathcal{G}$ the above guarantee on flow time implies that

$$\mathcal{C}(A, \mathcal{I}', s) \leq c\rho Opt(\mathcal{C}, \mathcal{I}', 1) \tag{7.1}$$

We now relate $\mathcal{I}'$ to $\mathcal{I}''$.

Fact 7.5 $\mathcal{C}(A, \mathcal{I}', s) = \mathcal{C}(A, \mathcal{I}'', 1)$

Proof. (of Fact 7.5:) This follows as jobs in $\mathcal{I}'$ are s times longer than jobs in $\mathcal{I}''$, and have exactly the same release times. So the schedule produced by A on $\mathcal{I}'$ using an s speed processor is indistinguishable by schedule produced by A on $\mathcal{I}''$ using a 1 speed processor. ■

Now, by definition of the algorithm B, executing the algorithm A on $\mathcal{I}''$ with a speed 1 processor is exactly the schedule produced by B on $\mathcal{I}$ using a 1 speed processor. So the completion times are identical. This implies that

$$\mathcal{C}(B, \mathcal{I}, 1) = \mathcal{C}(A, \mathcal{I}'', 1) \tag{7.2}$$

Now using Facts 7.4 and 7.5 and Equations 7.1 and 7.2 it follows that

$$\mathcal{C}(B, \mathcal{I}, 1) \leq c\rho sOpt(\mathcal{C}, \mathcal{I}, 1)$$

Thus we are done. ■

Weighted ℓ_p Norms of Completion Time

For $\mathcal{G}(x_1, \ldots, x_n) = (\sum_i w_i x_i^p)^{1/p}$, it is easily seen that the scalability property is satisfied. We now show that $\mathcal{G}$ is 2-good.

Lemma 7.6 $\mathcal{G}(x_1, \ldots, x_n) = (\sum_i w_i x_i^p)^{1/p}$ *is 2-good for all* $p \geq 1$.

Proof. Let S and S' be two schedules and let $f_1, \ldots, f_n$ (resp. $c_1, \ldots, c_n$) and $f'_1, \ldots, f'_n$ (resp. $c'_1, \ldots, c'_n$) be the flow times (resp. completion times) under S and S'.

We know that, $(\sum_i w_i f_i^p)^{1/p} \leq c(\sum_i w_i f'^p_i)^{1/p}$. The weighted ℓ_p norms of completion times under S (resp. under S') can be written as $(\sum_i w_i(f_i + r_i)^p)^{1/p}$ (resp. $(\sum_i w_i(f'_i + r_i)^p)^{1/p}$).

By convexity, we have that

$$\begin{aligned}
\sum_i w_i(f_i + r_i)^p &\leq \sum_i 2^{p-1} w_i(f_i^p + r_i^p) \\
&\leq c^p 2^{p-1} \sum_i w_i f_i^p + 2^p \sum_i w_i r_i^p \\
&\leq c^p 2^{p-1} \sum_i w_i(f'^p_i + r_i^p) \\
&\leq c^p 2^{p-1} \sum_i w_i({f'}_i + r_i)^p
\end{aligned}$$

Thus the result follows. ∎

Thus by Theorems 3.8, 3.10 and 7.3 and Lemma 7.6 we get that,

Corollary 7.7 *There exist $O(1)$-competitive clairvoyant and non-clairvoyant algorithms for minimizing the weighted ℓ_p norms of completion time.*

7.4 Concluding Remarks

The results obtained by using this technique are unlikely to be the tightest possible. For example, for the problem of minimizing the weighted completion time on a single machine, our technique only gives a 4-competitive algorithm while the best known deterministic algorithm achieves a competitive ratio of 2 [24]. Our results should be best viewed as providing a proof of existence of an $O(1)$ competitive algorithm for a completion time problem, if there exists any $O(1)$-speed , $O(1)$ competitive algorithm for the corresponding flow time problem.

Chapter 8

Possibilities for Improvement and Future Directions

8.1 Possibilities for Improvement

Two outstanding open problems that still remain in the area of flow time scheduling are

1. Is there an $O(1)$ competitive algorithm (or even an approximation algorithm) for minimizing the total weighted flow time on a single machine?
2. Is there an $O(1)$ approximation algorithm for minimizing the total flow time on multiple machines?

It is widely believed that there should be an $O(1)$ competitive online algorithm for minimizing weighted flow time. This belief is primarily based on two reasons. First, the inability to find an example that implies a non-constant lower bound on the competitive ratio. Second, that there exist two algorithms which have totally "different" guarantees, the first is the algorithm of Chekuri et al [23] that gives a bound of $O(\log^2 B)$, which is purely in terms of job sizes, irrespective of the weights involved. The second is our $O(\log W)$ competitive algorithm, the guarantee of which does not depend on job sizes at all. It seems that both these algorithms are not the right answers to the problem.

One approach that seems promising is based on the recent work of Becchetti et al [14]. They give an $O(1)$ competitive algorithm for minimizing total flow time in the model where the size of a job is not known exactly but only within an interval $[2^i, 2^{i+1})$ (formally this is known as semi-clairvoyant scheduling). The interesting property of this algorithm is that at any time only a constant fraction of the unfinished jobs are partially executed (unlike other algorithms like say *SRPT* or *SJF* where all jobs could be partially executed at some time). In fact, it satisfies a stronger property that if there are two or more jobs, then for every partially executed job there exists a unique larger job that is *total* i.e., not partially executed. If we could somehow extend this algorithm to the weighted case, then there is hope that this gives an $O(1)$ competitive algorithm by the following argument: First we can charge the weight of the partially executed jobs under the online algorithm to the total jobs, since this weight is only a constant factor of the weight of total jobs, we only need to worry about the weight in total jobs. Second, using some kind of work conservation argument, we can perhaps argue that the optimum algorithm too has at least a constant fraction of the weight contained in the unfinished *total* jobs under the online algorithm.

Our quasi-polynomial time approximation scheme for minimizing total flow time on $O(1)$ machines suggests that there should be a PTAS for this problem. The reason for the extra $\log n$ in the exponent of the running time of the algorithm presented in this thesis roughly comes for the following reason: We round jobs sizes such that they range from 1 to n^2. It seems that this cannot be avoided, in particular if we only have a constant range for the sizes then we seem to lose too much in the approximation, as there errors could build up over the n jobs. Thus we have $\Omega(\log n)$ job classes. Next, there does not seem a reasonable way to look at the job classes separately while constructing the schedule. This type of idea works for obtaining a PTAS for minimizing total stretch on a single machine [18] (essentially, since small jobs have a relatively larger weight than large jobs, it suffices to schedule the small jobs optimally before moving on to jobs of a higher size class). However, it is not clear if something similar could be done for flow time.

The algorithm *SETF* is $(1+\epsilon)$-speed, $O(\log^2 B/\epsilon^4)$-competitive for minimizing the total stretch non-clairvoyantly (Chapter 4, Theorem 4.1). It seems that the right answer should be a $(1+\epsilon)$-speed, $O(\log B)$-competitive algorithm. The main reason why we lose an additional factor of $\log B$ is that in our analysis is that while considering the contribution to stretch, we replace the size of a job by the amount of processing received by it (which could be much smaller). So, suppose if the input instance only had a single job of size B, then while the stretch would be 1, our analysis would yield $\Omega(\log B)$ (as each time interval $(2^i, 2^{i+1}]$ yields a constant amount to the total stretch). Since a non-clairvoyant algorithm cannot distinguish between a small job and a big one until it has given some service to both the jobs, it is not clear how to avoid this in the analysis.

8.2 Some Future Directions

Theoretical research in scheduling is closely tied with practice. Often new computer system architectures/technologies and new performance measures used to experimentally evaluate a system lead to new directions and problems in scheduling. For example, with the recent popularity of multicast and satellite broadcasting systems such as Direct-TV, broadcast scheduling has received a lot of theoretical interest in the last couple of years. As far as measures of performance are concerned, we believe that flow time will continue to be of fundamental importance. Some issues that will need to be addressed however, to make the theoretical results more relevant to practitioners seem to be the following:

1. *Understanding the trade-off between preemptions and flow time*: While it is known of non-preemptive algorithms are not good at all, it might be interesting to see if almost optimal flow times can be achieved via algorithms that requires "very" few preemptions. Some work in this direction has been in [28].

2. *Incomplete knowledge of jobs sizes*: Often in a real system, while job sizes may not be known exactly in advance, it is usually possible to get a rough idea of the job size by learning from past data etc. This naturally leads to a setting that lies between the clairvoyant and the non-clairvoyant model. An attempt at this has been recently made via the study of semi-clairvoyant scheduling [18, 14].

Bibliography

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

[2] N. Alon, Y. Azar, G. Woeginger, and T. Yadid. Approximation schemes for scheduling. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 493–500, 1997.

[3] A. Avidor, Y. Azar, and J. Sgall. Ancient and new algorithms for load balancing in the l_p norm. *Algorithmica*, 29(3):422–441, 2001.

[4] N. Avrahami and Y. Azar. Minimizing total flow time and completion time with immediate dispacthing. In *Proceedings of* 15^{th} *SPAA*, pages 11–18, 2003.

[5] B. Awerbuch, Y. Azar, E. Grove, M. Kao, P. Krishnan, and J. Vitter. Load balancing in the l_p norm. In *IEEE Symposium on Foundations of Computer Science*, 1995.

[6] 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.

[7] N. Bansal and K. Dhamdhere. Minimizing weighted flow time. In *Symposium on Discrete Algorithms SODA*, pages 508–516, 2003.

[8] N. Bansal, K. Dhamdhere, J. Konemann, and A. Sinha. Non-clairvoyant scheduling for minimizing mean slowdown. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 260–270, 2003.

[9] N. Bansal and M. Harchol-Balter. Analysis of SRPT scheduling: Investigating unfairness. In *ACM Sigmetrics*, pages 279–290, 2001.

[10] N. Bansal and K. Pruhs. Server scheduling in the weighted ℓ_p norm. accepted to appear in LATIN, 2004.

[11] N. Bansal and K. Pruhs. Server scheduling in the l_p norm: A rising tide lifts all boats. In *ACM Symposium on Theory of Computing (STOC)*, pages 242–250, 2003.

[12] Y. Bartal, S. Leonardi, A. Marchetti-Spaccamela, J. Sgall, and L. Stougie. Multiprocessor scheduling with rejection. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1996.

[13] 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.

[14] L. Becchetti, S. Leonardi, A. Marchetti-Spaccamela, and K. Pruhs. Semi-clairvoyant scheduling. In *Proc. of the 11th Annual European Symposium on Algorithms (ESA03)*, 2003.

[15] L. Becchetti, S. Leonardi, and S. Muthukrishnan. Scheduling to minimize average stretch without migration. In *Symposium on Discrete Algorithms*, pages 548–557, 2000.

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

[17] 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.

[18] M. Bender, S. Muthukrishnan, and R. Rajaraman. Improved algorithms for stretch scheduling. In *13th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2002.

[19] A. Borodin and R. El-Yaniv. *On-Line Computation and Competitive Analysis*. Cambridge University Press, 1998.

[20] Peter Brucker. *Scheduling Algorithms*. Springer Verlag, 2001.

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

[22] C. Chekuri, S. Khanna, and A. Kumar. Multi-processor scheduling to minimize ℓ_p norms of flow and stretch, 2003. unpublished manuscript.

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

[24] C. Chekuri, R. Motwani, B. Natarajan, and C. Stein. Approximation techniques for average completion time scheduling. *SIAM Journal on Computing*, 31(1):146–166, 2001.

[25] R. W. Conway, W. L. Maxwell, and L. W. Miller. *Theory of Scheduling*. Addison-Wesley Publishing Company, 1967.

[26] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Scheduling in Real-Time Systems*. John Wiley & Sons, 2002.

[27] M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in web servers. In *USENIX Symposium on Internet Technologies and Systems*, pages 243–254, 1999.

[28] J. Edmonds. Scheduling in the dark. *Theoretical Computer Science*, 235(1):109–141, 2000.

[29] D. Engels, D. Karger, S. Kolliopoulos, S. Sengupta, R. Uma, and J. Wein. Techniques for scheduling with rejection. *European Symposium on Algorithms*, pages 490–501, 1998.

[30] L. Epstein and J. Sgall. Approximation schemes for scheduling on uniformly related and identical parallel machines. In *European Symposium on Algorithms (ESA)*, pages 151–162, 1999.

[31] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP Completeness*. W.H. Freeman and Company, New York, 1979.

[32] M.X. Goemans, M. Queyranne, A.S. Schulz, M. Skutella, and Y. Wang. Single machine scheduling with release dates. *SIAM Journal on Discrete Mathematics*, 15:165–192, 2002.

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

[34] L. Hall, D. Shmoys, and J. Wein. Scheduling to minimize average completion time: Off-line and on-line algorithms. In *ACM-SIAM Symposium on Discrete Algorithms*, 1996.

[35] M. Harchol-Balter, M. Crovella, and S. Park. The case for srpt scheduling of web servers. Tech report, MIT-LCS-TR-767.

[36] G. Hardy, J. E. Littlewood, and G. Polya. *Inequalities*. Cambridge University Press, 1952.

[37] D. S. Hochbaum. *Approximation algorithms for NP-hard problems*. PWS Publishing Company, 1997.

[38] D. S. Hochbaum and D. B. Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *Journal of the ACM (JACM)*, 34(1):144–162, 1987.

[39] H. Hoogeveen, M. Skutella, and G. Woeginger. Preemptive scheduling with rejection. *European Symposium on Algorithms*, 2000.

[40] J. A. Hoogeveen, P. Schuurman, and G. J. Woeginger. Non-approximability results for scheduling problems with minsum criteria. In *Integer Programming and Combinatorial Optimization*, volume LNCS 1412, pages 353–366. Springer, 1998.

[41] E. Horowitz and S. Sahni. Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the ACM (JACM)*, 23(2):317–327, 1976.

[42] `http://httpd.apache.org/docs/`.

[43] `http://www.netcraft.com/survey/`.

[44] K. Jansen and L. Porkolab. Improved approximation schemes for scheduling unrelated parallel machin es. In *ACM symposium on Theory of computing*, pages 408–417, 1999.

[45] B. Kalyanasundaram and K. Pruhs. Minimizing flow time nonclairvoyantly. In *IEEE Symposium on Foundations of Computer Science*, pages 345–352, 1997.

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

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

[48] 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.

[49] D. Knuth. *The TeXbook*. Addison Wesley, 1986.

[50] J. Kurose and K. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison-Wesley, 2001.

[51] J. Kurose and K. Ross. *Computer networking: A top-down approach featuring the Internet*. Addison-Wesley, 2002.

[52] 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.

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

[54] J. Lenstra, D. Shmoys, and E. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46:259–271, 1990.

[55] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. In *ACM Symposium on Theory of Computing (STOC)*, pages 110–119, 1997.

[56] A. W. Marshall and I. Olkin. *Inequalities: Theory of Majorization and Its Applications*. Academic Press, 1979.

[57] N. Megow and A. Schulz. Scheduling to minimize average completion time revisited: Deterministic on-line algorithms. In *Proceedings of the First Workshop on Approximation and Online Algorithms (WAOA)*, 2003.

[58] T. Morton and D.W. Pentico. *Heuristic Scheduling Systems : With Applications to Production Systems and Project Management*. Interscience, 1993.

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

[60] 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.

[61] C. A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. In *ACM Symposium on Theory of Computing (STOC)*, pages 140–149, 1997.

[62] M. Pinedo. *Scheduling: Theory, Algorithms and Systems*. Prentice Hall, 1995.

[63] K. Pruhs, J. Sgall, and E. Torng. Online scheduling. Handbook on Scheduling: Algorithms, Models and Performance Analysis (to appear), CRC press.

[64] M. Queyranne and A. Schulz. Polyhedral approaches to machine scheduling, 1994. Preprint 408/1994, Dept. of Mathematics, Technical University of Berlin.

[65] B. Schroeder and M. Harchol-Balter. Web servers under overload: how scheduling can help. CMU technical report, CMU-CS-02-143.

[66] A. Schulz and M. Skutella. The power of alpha-points in preemptive single machine scheduling. *Journal of Scheduling*, 5:121–133, 2002.

[67] P. Schuurman and G. J. Woeginger. Polynomial time approximation algorithms for machine scheduling: Ten open problems. *Journal of Scheduling*, 2:203 – 213, 1999.

[68] Steven S. Seiden. Preemptive multiprocessor scheduling with rejection. *Theoretical Computer Science*, 262(1–2):437–458, 2001.

[69] 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.

[70] D. Shmoys, J. Wein, and D. P. Williamson. Scheduling parallel machines on-line. *SIAM Journal on Computing*, 24(6):1313–1331, 1995.

[71] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.

[72] A. Tanenbaum. *Operating systems: design and implementation*. Prentice-Hall, 2001.

[73] K. Thompson. Unix implementation. *The Bell System Technical Journal*, 57(6):1931–1946, 1978.

[74] V. Vazirani. *Approximation Algorithms*. Springer-Verlag, 2001.

[75] 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.