SRPT Optimally Utilizes Faster Machines to Minimize Flow Time

Jason McCullough*and Eric Torng†
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824
torng@msu.edu

Abstract

We analyze the shortest remaining processing time (SRPT) algorithm with respect to the problem of scheduling n jobs with release times on m identical machines to minimize total flow time. It is known that SRPT is optimal if $m = 1$ but that SRPT has a worst-case approximation ratio of $\Theta(\min(\log n/m, \log \Delta))$ for this problem, where Δ is the ratio of the length of the longest job divided by the length of the shortest job. It has previously been shown that SRPT is able to use faster machines to produce a schedule *as good as* an optimal algorithm using slower machines. We now show that SRPT *optimally* uses these faster machines with respect to the worst-case approximation ratio. That is, if SRPT is given machines that are $s \geq 2 - 1/m$ times as fast as those used by an optimal algorithm, SRPT's flow time is at least **s times smaller** than the flow time incurred by the optimal algorithm. Clearly no algorithm can offer a better worst-case guarantee, and we show that existing algorithms with similar performance guarantees to SRPT without resource augmentation do not optimally use extra resources.

1 Introduction

In this paper, we consider the problem of scheduling m identical machines to minimize *total flow time*. In more detail, we are given m identical machines and an input instance I, which is a collection of n independent jobs $\{J_1, J_2, \ldots, J_n\}$. Each job J_j has a release date r_j and a size or length p_j. Note that size is commonly referred to as processing time, but since we will consider machines that run at different speeds, length or size is more appropriate. A job can run on only one machine at a time, and a machine can process only one job at a time. We denote the completion time of job J_j in schedule S by C_j^S and drop the superscript when the schedule is understood from context. The *flow time* of a job in schedule S is $F_j^S \equiv C_j^S - r_j$. The *idle time* or *delay* of a job in schedule S is $D_j^S \equiv F_j^s - p_j \equiv C_j^s - r_j - p_j$. The *total flow time* of schedule S is $\sum_j F_j^S$. We consider a preemptive and migratory scheduling model where a job may be interrupted and subsequently resumed on any machine with no penalty. In the classic notation of Lenstra *et al.*, this is the $P \mid pmtm \mid \sum_j F_j$ problem. Instead of focusing on total flow time, we could equivalently consider the minimization of *average flow time* $\frac{1}{n}\sum F_j$. Furthermore, total flow time is minimized when we minimize *total completion time*, $\sum_j C_j^S$, or *total idle time*, $\sum_j D_j^S$.

We focus our attention on the Shortest Remaining Processing Time (SRPT) algorithm that, at any time, schedules the m jobs with shortest remaining length (processing time) breaking ties arbitrarily. SRPT is an *online* scheduling algorithm. An online scheduling algorithm must construct the schedule up to time t without any prior knowledge of jobs that become available at time t or later. When a job arrives, however, we assume that all other relevant information about the job is known. In constrast, an *offline* scheduling algorithm has full knowledge of the input instance when constructing a schedule.

When the number of machines $m = 1$, SRPT is known to be an optimal algorithm for this problem. When $m \geq 2$, this problem is known to be NP-complete. Leonardi and Raz showed that the SRPT algorithm has a worst-case approximation ratio of $\Theta(\min(\log n/m, \log \Delta))$ for this problem, where Δ is the ratio of the length of the longest job divided by the length of the shortest job [10]. They also showed that $\Theta(\min(\log n/m, \log \Delta))$ is the best possible approximation ratio for any deterministic or randomized online algorithm. A simpler analysis of SRPT is given in [9]. No offline algorithm with a superior approximation ratio is known for this problem.

Kalyanasundaram and Pruhs popularized the us-

*Supported in part by NSF grant CCR 9701679, an MSU professorial assistantship, and an MSU summer research internship.

†Supported in part by NSF grants CCR 9701679 and CCR 0105283.

age of resource augmentation as a method for analyzing online algorithms, in particular online scheduling algorithms [8]. Using this technique, we compare the performance of an online algorithm to an offline algorithm when the online algorithm is given extra resources in the form of faster machines or extra machines. In this paper, we ignore extra machines and consider only faster machines as well as *stretched* input instances, a concept related to faster machines introduced in Phillips *et al.* [11]. A job with length p_j takes p_j/s time to complete if run on a speed-s machine.

We use the terminology introduced by Phillips *et al.* Let I be an instance of an m machine minimization scheduling problem with optimal solution value V. An *s-speed ρ-competitive algorithm* finds a solution of value at worst ρV using m speed-s machines. For any input instance I, define I^s to be the *s-stretched* input instance where job J_j has release time sr_j instead of r_j. An s-stretch ρ-competitive algorithm finds a solution to I^s of value at worst ρV using m speed-1 machines. The relationship between faster machines and stretched input instances is captured by the following speed-stretch theorem from Phillips *et al.*

THEOREM 1.1. *Any s-speed ρ-competitive algorithm is also an s-stretch ρs-competitive algorithm when working with the flow time metric.*

Note, the reverse holds as well. That is, any s-stretch ρ-competitive algorithm is also an s-speed ρ/s-competitive algorithm.

1.1 Our contributions and related work Our primary result is that SRPT *optimally* uses faster machines. That is, if SRPT is given speed-s machines where $s \geq 2 - 1/m$, then SRPT incurs a total flow time that is at least s times smaller than that incurred by the optimal algorithm using speed-1 machines. More formally, SRPT is an s-speed $1/s$-competitive algorithm for minimizing total flow time when $s \geq 2 - 1/m$. No algorithm can use faster machines to get a better worst-case guarantee as can be seen by considering an input instance consisting of a single job. This improves upon the result in Phillips *et al.* where they proved that SRPT is an s-speed 1-competitive algorithm for this problem when $s \geq 2 - 1/m$ [11]. In contrast, we also show that existing algorithms with similar performance guarantees to SRPT without extra resources are not s-speed $1/s$-competitive algorithms for any s. This includes the non-migratory algorithms developed by Awerbuch, Azar, Leonardi, and Regev [3], Chekuri, Khanna, and Zhu [5], and and Avrahami and Azar [2]. This offers some evidence in favor of SRPT for this problem.

In addition, we hope that several of the concepts and techniques used in this paper including profiles, SRPT charging, and stretched input instances may be helpful in proving other results concerning flow time and weighted flow time. Note, Anderson and Potts [1] used the concept of a double problem to help prove a result regarding minimizing total completion time in a nonpreemptive uniprocessor environment. In their double problem, they multiply not only release times but also processing times by a factor of 2 to create a related input instance.

Resource augmentation has been used to study the problem of minimizing flow time in a *nonclairvoyant uniprocessor* environment where the algorithm has no knowledge of p_j until job J_j completes [8, 6, 7]. Edmonds also shows that the Round Robin algorithm is an s-speed $O(1)$-competitive algorithm for the parallel machine problem for $s \geq 2$, but RR is not s-speed 1-competitive for $s < 4$ and is at best s-speed $2/s$-competitive for $s \geq 4$ for a more general problem setting [7]. More recently, Chekuri, Khanna, and Kumar have shown that the algorithm of Avrahamai and Azar is a $(1+\epsilon)$-speed $O(1/\epsilon)$-competitive algorithm for this problem (and others) [4]. This is an important result as it shows that with modest resource augmentation, a constant competitive ratio is achievable. However, as noted earlier, we can show that this algorithm does not optimally use extra resources as SRPT does. It is likely that SRPT is also a $(1+\epsilon)$-speed $O(1/\epsilon)$-competitive algorithm for minimizing total flow time, but its analysis is likely to be more complex.

The rest of this paper is organized as follows. In section 2, we first show that several algorithms are not s-speed $1/s$-competitive algorithms for any $s > 1$. In section 3, we first give an intuitive overview of the proof. In section 4, we introduce some building blocks for the proof such as profiles, partial schedules, canonical schedules, and an idle time accounting scheme we name *SRPT charging*. In section 5, we introduce, for analysis purposes only, an algorithm we name *Relaxed SRPT (RSRPT)*. We then show that RSRPT incurs no more idle time than the optimal algorithm and that SRPT on a stretched input instance incurs no more idle time than RSRPT on the original input instance. We conclude with a brief discussion of open problems.

2 Bounds on other algorithms

The key idea in algorithms that eliminate migration is the idea of classifying jobs by size [3, 5, 2]. In [3], jobs are classified as follows: a job j whose remaining processing time at time t is in $[2^k, 2^{k+1})$ is in class k for $-\infty < k < \infty$ at time t. Note that jobs change classes as they execute. In [5] and [2], a job j whose *initial* processing time is in $[2^k, 2^{k+1})$ is in class k for

$-\infty < k < \infty$ for all times t after its release up until its completion. Note, 2 is used to determine classes, but 2 could be c for any constant $c > 1$. In [5], they optimize their algorithm by identifying the best possible constant c.

The algorithms of [3, 5] use the following data structures to organize available jobs at any time. Jobs not yet assigned to any machine are stored in a central pool. Jobs assigned to machine k are stored on a stack for machine k. The algorithms of [3, 5] schedule jobs as follows. Each machine processes the job at the top of its stack. When a new job arrives, if there is any machine k that is idle or currently processing a job of a higher class than the new job, the new job is pushed onto machine k's stack and machine k begins processing the new job. If multiple machines satisfy the above criteria, the job is assigned to any of one of them. Otherwise, the job enters the central pool. When a job is completed on any machine k, machine k compares the class of the job on top of its stack (if such a job exists) with the minimum class of any job in the central pool. If the minimum in the pool is smaller than the class of the job on top of its stack, then any job in the pool of that minimum class is pushed onto machine k's stack. Machine k then begins processing the job on top of its stack. In [5], they also define an algorithm where migration is allowed so that when a job completes on machine k, machine k looks for the smallest class job on other machines' stacks in addition to the central pool.

We first show that the algorithms of [3, 5] cannot be s-speed $1/s$-competitive algorithms for this problem. We use A to denote any implementation of the non-migratory algorithms of [3, 5].

THEOREM 2.1. *As $m \to \infty$, A is at best an s-speed $\frac{3+\sqrt{13}}{1+\sqrt{13}}\frac{1}{s}$-competitive algorithm for any speed $s \geq 1$ and for any constant $c \geq 1$ used to define the classes of jobs in A.*

We prove this by considering two different example input instances, one that is more effective for small c, and one that is more effective for large c.

LEMMA 2.1. *A is at best an s-speed $\frac{2c+1}{c+2}\frac{1}{s}$-competitive algorithm for any $c \geq 1$ and any $s \geq 1$.*

Proof. Consider the following input instance. Suppose m jobs of size $c-\epsilon$ where $\epsilon > 0$ arrive at time 0, and m jobs of size 1 arrive at time $\delta > 0$. All $2m$ jobs belong to the same class 0 at time δ since their initial and remaining sizes at time δ lie in the range $[c^0, c^1)$. A will process the jobs of size $c-\epsilon$ first on each machine before processing the jobs of size 1 on each machine resulting in a total flow time of $(m/s)(2c - 2\epsilon + 1)$. The optimal strategy is to preempt the jobs of size $c-\epsilon$ for the jobs of size 1 resulting in a total flow time of $m(c-\epsilon+2)$. Since ϵ and δ can be made arbitrarily small, the result follows. This result actually holds for migratory versions of the above algorithms as well.

LEMMA 2.2. *A is at best an s-speed $\left(\frac{c}{c-1} - \frac{m}{c^m-1}\right)\frac{1}{s}$-competitive algorithm for any $c \geq 1$, $m \geq 2$, and $s \geq 1$. This asymptotically approaches $\frac{c}{c-1}\frac{1}{s}$ as $m \to \infty$.*

Proof. Consider the following input instance. We release $m-1$ jobs of size ϵ at time 0. A will assign each of these jobs to its own machine. Then, we release a sequence of m jobs at unique times in the interval $(0, \epsilon)$. The jobs released are of size c^k for $0 \leq k \leq m-1$, and the jobs are released in decreasing order of size. A will assign each of these jobs to the machine that did not schedule any job of size ϵ. Thus, A will incur a total flow time of $(1/s)\sum_{k=0}^{m-1}(m-k)c^m + \epsilon(m-1)/s = (1/s)[(c^{m+1} - c)/(c-1)^2 - m/(c-1) + (m-1)\epsilon]$. On the other hand, the m larger jobs could be assigned to individual machines for a total flow time of $\sum_{k=0}^{m-1} c^k = (c^m - 1)/(c-1) + 2(m-1)\epsilon$. As ϵ can be made arbitrarily small, the result follows.

The proof of Theorem 2.1 is completed by finding the value of c where $c/(c-1) = (2c+1)/(c+2)$, and this occurs when $c = (3+\sqrt{13})/2$. The lower bound on the competitive ratio then evaluates to $(3+\sqrt{13})/(1+\sqrt{13}) \approx 1.43$.

Similar arguments can also be developed to show that the algorithms of [2] and migratory versions of [3, 5] are not s-speed $1/s$-competitive.

3 Proof overview

3.1 Bad example for SRPT Before we describe the proof, it is helpful to review an example input instance for two machines that causes problems for SRPT without resource augmentation. Suppose 3 jobs are released at time 0 with lengths 1, 1, and 2. An optimal offline algorithm (Opt) will execute the job of length 2 on one machine from time 0 to time 2 while executing one of the jobs of size 1 on the second machine from time 0 to time 1 and the other job of size 1 on the second machine from time 1 to time 2. Thus, all jobs released at time 0 are completed by time 2. SRPT, on the other hand, will schedule the two jobs of size 1 on the two machines from time 0 to time 1 and the job of size 2 on either machine from time 1 to time 2. Thus, at time 2, SRPT has not "kept up with" Opt; in particular, SRPT completes less work in interval $[0, 2]$ than Opt does. (We will define a formal notion of "keeping up" in the building blocks section.) This pattern can now

be repeated with new jobs released at time 2 and so on to cause SRPT to have an arbitrarily larger flow time than Opt.

3.2 Proof outline SRPT with resource augmentation overcomes the issue in the following manner. The first step in our proof is to focus on SRPT with stretched input instances rather than faster machines. Based on the speed-stretch theorem cited earlier, we can prove our result if we show that SRPT is an s-stretch 1-competitive algorithm for $s \geq 2 - 1/m$.

The next step is to break the input instance into intervals as defined by the release times of input I. Let I_i be the interval defined by the i^{th} and $i+1^{st}$ release times. We can show that SRPT at the end of stretched interval I_i^s is at least "keeping up with" Opt at the end of interval I_i. If we could show that the flow time incurred by SRPT on the stretched interval I_i^s is at most the flow time incurred by Opt on the interval I_i, we would be done. However, this often is not true because the stretched interval I_i^s is s times longer than interval I_i.

Instead, we define for analysis purposes only a Relaxed SRPT (RSRPT) algorithm. We then show that RSRPT "keeps up with" Opt at the end of each interval I_i and that the flow time incurred by RSRPT on each interval I_i is at most the flow time incurred by Opt on I_i. The key idea behind RSRPT is that it can produce illegal schedules where some machines may run for more than the total time in an interval. This will be compensated by some machines running for less total time in the interval. To illustrate, consider the bad example for SRPT cited earlier. The RSRPT "schedule" for the interval from time 0 to time 2 will be jagged. Machine one will run for three units of time processing one of the jobs of size 1 from time 0 to time 1 and the job of size 2 from time 1 to time 3. The other machine will run for only one unit of time processing the other job of size 1 from time 0 to time 1. It may be possible in extreme cases for RSRPT to allocate all of its processing in some interval to one machine and one job.

Given that RSRPT may produce illegal jagged schedules for a given interval of time, we need to define a way to compute a flow time cost for RSRPT during this interval. We develop a charging scheme that we name SRPT charging that computes the idle time incurred by each job in the jagged schedule assuming we processed all jobs to completion using SRPT and that no new jobs arrived. For example, the job of size 1 on the machine one would incur an idle time of 1 since the job of size 2 was idle for one unit of time waiting for the first job to finish. The other job of size 1 would incur an idle time of 0 since no jobs would be scheduled after it on its machine. Likewise, the job of size 2 incurs an idle time of 0. To estimate Opt's flow time (or idle time) for each interval I_i, we develop a notion of a canonical schedule for each such interval and argue that Opt's idle time cost cannot be smaller than the idle time cost of the canonical schedule as computed by SRPT charging. Puting this together, we conclude that the total flow time of $RSRPT(I)$ is no larger than the total flow time of $Opt(I)$.

We now need to show that the total flow time incurred by RSRPT for I is at least the total flow time incurred by SRPT for I^s. To prove this, we define a notion of *containment* and then ensure that at each release time, $SRPT(I^s)$ is always contained within $RSRPT(I)$. Essentially, this containment property means that the number of jobs with remaining length larger than the remaining length for a given job J_j in $RSRPT(I)$ at time t is at least as many as the number of jobs with remaining length larger than the remaining length for the equivalent job in $SRPT(I^s)$ at time st. This containment property then allows us to argue on a job by job basis that the idle time incurred by each job in $RSRPT(I)$ is at least as large as the idle time incurred by the equivalent job in $SRPT(I^s)$ when we apply SRPT charging to both schedules. Thus, by transitivity, we are able to conclude that the idle time and thus total flow time of $SRPT(I^s)$ is no larger than the idle time and total flow time of $Opt(I)$. We now proceed to a more detailed description of this proof.

4 Building blocks

4.1 Interval and partial schedule notation For any input instance I, let $r(I)$ denote the number of distinct release times of jobs in I, and $r_i(I)$ denote the i^{th} release time in I for $1 \leq i \leq r(I)$. When the input instance I is unambiguous, we use the notation r_i for $r_i(I)$. We use these release times to define time intervals as follows. Time interval I_i is defined to be $[r_i(I), r_{i+1}(I))$ for $1 \leq i \leq r(I) - 1$. Time interval $I_{r(I)}$ is defined to be $[r_{r(I)}, \infty)$. We use I_{i-} to denote the time interval $[r_1(I), r_i(I))$ for $1 \leq i \leq r(I)$. For $1 \leq i \leq r(I) - 1$, we use $|I_i|$ to denote the length of interval I_i. For any schedule $S(I)$ and $1 \leq i \leq r(I)$, we define $S(I_i)$ and $S(I_{i-})$ to be the partial schedules of $S(I)$ for intervals I_i and I_{i-}, respectively.

For example, consider the input instance on two machines where two jobs of size 1 and one job of size 2 are released at time 0 and two jobs of size 3 are released at time 2. Then $r(I) = 2$, $r_1(I) = 0$, $r_2(I) = 2$, $I_1 = [0, 2)$, $I_{1-} = [0, 0)$ which is empty, $I_2 = [2, \infty)$, and $I_{2-} = I_1 = [0, 2)$. The partial schedule $SRPT(I_1)$ (which is also partial schedule $SRPT(I_{2-})$) is to run

the two jobs of size 1 from time 0 to time 1 and the one job of size 2 from time 1 to 2 on one of the machines. The partial schedule $SRPT(I_2)$ is to run the job of size 2 from time 2 to 3, one of the jobs of size 3 on the same machine from time 3 to time 6, and the other job of size 3 on the other machine from time 2 to time 5.

4.2 Profiles We will compare different schedules for an input instance I at the $r(I)$ different release times of I. To facilitate this comparison, we introduce the notion of a profile of a schedule and the notion of one profile being larger than another. This is the formalization of a schedule "keeping up with" another schedule.

DEFINITION 4.1. *Let I be any input instance, and let $S(I)$ be a legal schedule for I. We define the profile of schedule $S(I)$ at release time r_i for $1 \leq i \leq r(I)$, denoted $S(I,i)$, to be the non-decreasing vector of remaining lengths of all jobs that were released up to but not including time r_i. We use $S[I,i]$ to denote the profile that includes jobs released at time r_i. We define $|S(I,i)|$ and $|S[I,i]|$ to be the number of elements in profiles $S(I,i)$ and $S[I,i]$, respectively.*

DEFINITION 4.2. *We define $S(I,i)[j]$ and $S[I,i][j]$ to be the jobs with the jth smallest remaining length in profiles $S(I,i)$ and $S[I,i]$, respectively. If two jobs tie for the jth smallest remaining length, $S(I,i)[j]$ (or $S[I,i][j]$) is the one that received more processing time in partial schedule $S(I_{i-1})$. If there is still a tie in amount of processing time received in $S(I_{i-1})$, ties are broken arbitrarily. We overload notation and also use $S(I,i)[j]$ and $S[I,i][j]$ to denote that job's remaining length at release time r_i.*

We can define profiles at any time, not just release times, but we will only use profiles at release times in this paper. When working with general profiles independent of a specific schedule or input instance, we will use the notation P or P_i to denote a profile and the notation $|P|$ or $|P_i|$ to denote that profile's number of elements.

DEFINITION 4.3. *We say that a profile P_1 is* smaller than *another profile P_2 if the following conditions hold:*

1. *$|P_1| \leq |P_2|$.*
2. *For $1 \leq i \leq |P_1|$, the sum of the first i elements of profile P_1 is no larger than the sum of the first i elements of profile P_2.*

We denote this by writing $P_1 \leq P_2$.

DEFINITION 4.4. *We say that a profile P_1 is* contained *by another profile P_2 if the following conditions hold:*

1. *$|P_1| \leq |P_2|$.*
2. *For $1 \leq i \leq |P_1|$, the ith element of profile P_1 is no larger than the ith element of profile P_2.*

We denote this by writing $P_1 \subseteq P_2$.

Concepts similar to a profile have been used in many other papers analyzing SRPT and other algorithms for minimizing total flow time and other objective functions. One key point about our definition of profiles is that we include the jobs with remaining length 0 in the vector. For example, consider a uniprocessor environment and an input instance I where jobs of size 1, 1, and 2 are released at time 0 and a job of size 1 is released at time 2. Suppose schedule $S(I)$ schedules the job of size 2 on the single machine from time 0 to time 2. Then $SRPT(I,2) = \langle 0,0,2\rangle \leq S(I,2) = \langle 0,1,1\rangle$. However, it is not true that $SRPT(I,2) \subseteq S(I,2)$. Furthermore, $SRPT(I,2)[2] = 0$ while $S(I,2)[2] = 1$.

We will sometimes use a profile P as an input instance to this problem with a single release time. Specifically, profile P is an input instance with $|P|$ jobs, the ith smallest job of the instance has the size of the ith smallest element of P, and all jobs are assumed to be released simultaneously. Sometimes, for convenience, we will assume that the jobs of size 0 do not exist and remove them from the instance. Typically, though, this is not necessary.

We now list a few easy to prove observations about profiles.

FACT 4.1. *Let P_1 and P_2 be two arbitrary profiles such that $P_1 \leq P_2$. Then the sum of all remaining lengths in P_1 is no larger than the sum of all remaining lengths in P_2.*

Proof. This follows from the definition of $P_1 \leq P_2$.

FACT 4.2. *Let P_1 and P_2 be two profiles such that $P_1 \leq P_2$ ($P_1 \subseteq P_2$, respectively). If we add a job of size x to both P_1 and P_2 to create P_1' and P_2', then $P_1' \leq P_2'$ ($P_1' \subseteq P_2'$, respectively).*

COROLLARY 4.1. *Let I be any input instance. For any $1 \leq i \leq r(I)$ and $s_1, s_2 \geq 1$, and any schedules S_1 and S_2 such that $S_1(I^{s_1},i) \leq S_2(I^{s_2},i)$ ($S_1(I^{s_1},i) \subseteq S_2(I^{s_2},i)$, respectively), then $S_1[I^{s_1},i] \leq S_2[I^{s_2},i]$ ($S_1[I^{s_1},i] \subseteq S_2[I^{s_2},i]$, respectively).*

Continuing the example started above, this implies that $SRPT[I,2] = \langle 0,0,1,2\rangle \leq S[I,2] = \langle 0,1,1,1\rangle$.

4.3 Building blocks from previous work There are two critical ideas we need to borrow from the paper of Phillips *et al.* The first is the speed-stretch theorem

cited earlier. Based on this theorem, we can prove our result if we show that SRPT is an s-stretch 1-competitive algorithm for $s \geq 2 - 1/m$. Thus, we consider SRPT with stretched input instances rather than faster machines throughout most of this paper. Also, as noted earlier, total flow time is minimized exactly when total idle time is minimized. Thus, our goal is to show that SRPT with stretched input instances incurs no more idle time than the optimal algorithm does with the unstretched input instance.

The second critical idea is a generalization of the proof that SRPT is a $(2-1/m)$-speed 1-competitive algorithm. That proof worked in two steps. In the first step, Phillips *et al.* showed that any busy scheduling algorithm, when given speed-$(2-1/m)$ machines, performs at least as much work by any given time as any other schedule on any input instance. They extended this to also conclude that SRPT, given speed-$(2-1/m)$ machines, completes at least as many jobs by any given time as any other schedule on any input instance. We generalize Phillips *et al.*'s result to show that SRPT on I^s is at least "keeping up with" Opt on I.

COROLLARY 4.2. *Consider any input instance I, any legal speed-1 schedule $S(I)$, and any i $1 \leq i \leq r(I)$. If $s \geq 2 - 1/m$, then $SRPT(I^s, i) \leq S(I, i)$.*

Proof. The proof is essentially identical to that of Phillips *et al.* It also holds for any time t, not just release times r_i.

4.4 Canonical schedules We now define the notion of a canonical partial schedule $S(I_i)$.

DEFINITION 4.5. *For any input instance I and $1 \leq i \leq r(I)$, partial schedule $S(I_i)$ is* canonical *if it has the following properties:*

1. *No job that is finished in $S(I_i)$ had a remaining length at time r_i larger than any job that is not completed in $S(I_i)$.*
2. *Each machine processes at most one job that is not completed in $S(I_i)$.*
3. *Suppose jobs J_j and J_k are processed but not completed in $S(I_i)$ and that job J_j receives more processing time in $S(I_i)$ than J_k. Then job J_j had a smaller remaining length at time r_i than job J_k did.*
4. *On any machine, no unfinished job is processed prior to any finished job in $S(I_i)$.*

LEMMA 4.1. *Any legal partial schedule $S(I_i)$ for interval I_i can be converted into a canonical schedule $S'(I_i)$ for interval I_i such that $S'(I_i) \leq S(I)$ and $S'(I_i)$ incurs no more idle time than $S(I_i)$ does.*

Proofsketch. The proof is to use job swapping arguments. We first swap jobs to ensure that there are no finished jobs in interval I_i that are longer (at time r_i) than any unfinished jobs in interval I_i. We then swap jobs to ensure that the unfinished jobs with smallest length are processed whenver there is a choice of unfinished jobs to work on and that these unfinished jobs are processed on a single machine. Finally, we swap jobs to ensure that finished jobs are processed before unfinished jobs on the same machine.

4.5 SRPT charging We now define a method for computing a lower bound on the amount of idle time incurred by a canonical partial schedule. We will also use this to determine the amount of idle time incurred by RSRPT (still to be defined) on a partial schedule. We term this method *SRPT Charging* as we charge the canonical partial schedule as if it ran its jobs as SRPT would. This may produce an illegal partial schedule in some cases. For example, suppose the canonical two machine partial schedule for jobs of size 1, 1, 2, 3, 4, 5 in the interval $[0, 2]$ is to run the two jobs of size 1 on one machine and one job of size 2 on the second machine. If we assume that SRPT runs the exact same jobs, we get instead that the two jobs of size 1 run simultaneously from time 0 to time 1 and the job of size 2 is processed from time 1 to time 3. This leaves a jagged pattern with one machine finishing at time 1 and the other finishing at time 3 rather than both finishing at time 2.

In order to charge idle times, we use the following ordering on jobs. We number the jobs in non-decreasing order of remaining length at the beginning of the interval, breaking ties arbitrarily for completed jobs. For jobs that are not completed in this partial schedule, we break ties by the amount each job is processed. If one job is processed more in the partial schedule, it receives a smaller number. Otherwise, ties are broken arbitrarily. Let k be the total number of jobs with non-zero remaining lengths at the beginning of the interval.

We then charge idle time as follows. Suppose job numbered j is processed for e_j time units in this interval. We then say that jobs $j+m$, $j+2m$, ... incur e_j units of idle time and we will charge these incurred idle times to job j.

For our example, the first job of size 1 is charged 1 unit of idle time each for the jobs of size 2 and 4. The second job of size 1 is charged 1 unit of idle time each for the jobs of size 3 and 5. The job of size 2 is charged 2 units of idle time for the job of size 4. Thus, a total of 6 units of idle time are incurred. In the actual canonical schedule, the second job of size 1 incurs one unit of idle time, and each of the jobs of size 3, 4 and 5 each incur two units of idle time for a total of 7 units

of idle time. Thus, the idle time incurred by applying SRPT charging is at most (in this case smaller than) the actual idle time incurred. We now prove that this is always the case for any canonical schedule.

LEMMA 4.2. *The actual amount of idle time incurred by any canonical partial schedule is at least the amount of idle time incurred by applying SRPT charging to that canonical schedule.*

Proofsketch. Let J denote the set of jobs at the beginning of the interval of I_i. Let $C(I_i)$ be the canonical partial schedule under consideration. Let us ignore all jobs that arrive after time r_i; that is, we assume the single release time case where our goal is to schedule the jobs in J in order to minimize total idle time. It is known that SRPT minimizes total flow time and total idle time in cases where there is a single release time. Thus $SRPT(J)$ incurs no more idle time than that incurred by $C(I_i)$ plus any legal schedule to complete all the jobs in J after interval I_i ends.

We can apply SRPT charging to $SRPT(J)$ to exactly compute the total idle time of $SRPT(J)$. We can decompose this cost into two components: SRPT charging of canonical partial schedule $C(I_i)$ and SRPT charging of the remainder of $SRPT(J)$. Since $SRPT(J)$ is optimal, this total idle time cost must be no more than the true idle time cost of $C(I_i)$ plus the minimum idle time cost for completing the remaining jobs after the end of the interval. By the property of being a canonical schedule, the order of remaining processing times of jobs not completed in partial schedule $C(I_i)$ is identical to their order at the beginning of the interval. Thus, a minimum idle time cost for completing the remaining jobs after the end of the interval is to apply SRPT to the remaining jobs. Thus, the minimum idle time cost for completing these jobs is identical to the cost of SRPT charging the remaining jobs after the end of the interval. This means that SRPT charging of $C(I)$ must be no larger than the actual idle time cost incurred by $C(I)$.

5 Proof of main result

We now prove the main result. Let I be any input instance. Consider any optimal schedule $Opt(I)$ for input instance I. For each interval I_i, we lower bound the idle time incurred by partial schedule $Opt(I_i)$ by working with the idle time incurred by the corresponding canonical partial schedule $C(I_i)$. We would like to show that the total idle time incurred by each canonical partial schedule $C(I_i)$ is at least the idle time incurred by $SRPT(I_i^s)$ for $s \geq 2 - 1/m$. However, this may not be true as I_i^s is s times longer than I_i.

5.1 Relaxed SRPT (RSRPT) In order to be able to implement such a comparison, we define for analysis purposes only the Relaxed Shortest Remaining Processing Time (RSRPT) schedule. We will show that the idle time incurred by canonical partial schedule $C(I_i)$ is at least the idle time incurred by $RSRPT(I_i)$ and that the total idle time incurred by $RSRPT(I)$ is at least the total idle time incurred by $SRPT(I^s)$.

We construct $RSRPT(I)$ interval by interval. One factor driving the formation of $RSRPT(I_i)$ will be the schedule $Opt(I_{i-})$ concatenated with schedule $C(I_i)$. We will use the notation $OptC(I_i)$ to denote this concatenated schedule. The other factor we will use to construct $RSRPT(I_i)$ is $SRPT(I^s_{(i+1)-})$. Any of the partial schedules $RSRPT(I_i)$ may be illegal, but we still concatenate them together to form $RSRPT(I)$. To determine the idle time incurred by these potentially illegal partial schedules, we use SRPT charging. It is important to note that $RSRPT$ is dependent on the stretch factor s. Technically, we should include s as a parameter in the definition of $RSRPT(I)$, but we choose to omit it to simplify notation.

Before we construct $RSRPT(I_i)$, we define the following quantities based on $OptC(I_i)$. Let $OptC(I, i+1)$ denote the profile at the end of $OptC(I_i)$ not including jobs released at time r_{i+1}. Let k be the number of jobs with zero remaining length (some of these jobs may have been completed in $Opt(I_{i-})$). Finally, let jobs $OptC(I, i+1)[k+1]$ through $OptC(I, i+1)[k+l]$ denote the jobs that received some processing in $C(I_i)$ but are not complete by time r_{i+1}, and let e_j denote the processing time received by job $OptC(I, i+1)[k+j]$.

Now consider profile $RSRPT[I, i]$, the vector of remaining lengths of jobs to be processed by RSRPT in interval I_i. We first give enough processing time to complete jobs $RSRPT[I, i][j]$ for $1 \leq j \leq k$. That is, RSRPT must complete at least k jobs by the end of $RSRPT(I_i)$. Note, some of these jobs may start with zero remaining length meaning they were completed in $RSRPT(I_{i-})$. Now for $1 \leq j \leq l$, we assign the minimum of e_j and the time required to complete job $RSRPT[I, i][k+j]$ to this job in $RSRPT(I_i)$. That is, either job $RSRPT[I, i][k+j]$ is completed or it receives e_j processing time. Let q be the largest value of j for $0 \leq j \leq l$ such that $RSRPT[I, i][k+j]$ is completed by the above assignment.

Let X denote the total amount of processing $C(I_i)$ devoted to jobs $OptC(I, i+1)[j]$ for $1 \leq j \leq k+q$ minus the total amount of processing $RSRPT(I_i)$ devoted to jobs $RSRPT[I, i][j]$. We will prove later that X must be nonnegative. The remainder of the construction of $RSRPT(I_i)$ is characterized by the following pseudocode. Intuitively, we apply the excess

processing time to the remaining jobs with priority given to the shortest remaining jobs. The limits on the application of processing to a job are either the amount needed to complete it or the amount received by the corresponding job in the profile $SRPT(I^s, i+1)$. The second limit is given to ensure that $SRPT(I^s, i+1)$ is contained in $RSRPT(I, i+1)$.

- Let $y = k + q + 1$.
- Define $X_y = X$.
- While $X_y > 0$ {
  - Add processing time to $RSRPT[I, i][y]$ until
    - $RSRPT[I, i][y]$ is complete
    - We have added X_y processing time,
    - or $RSRPT(I, i+1)[y] = SRPT(I^s, i+1)[y]$.
  - Let a_y be the amount of processing time we added to this job.
  - Let $X_{y+1} = X_y - a_y$.
  - Increment y by 1
- }

5.2 RSRPT incurs at least as much idle time as stretched SRPT The key to proving that RSRPT incurs at least as much idle time as SRPT does on a stretched input instance is showing that the profiles created by SRPT on I^s are always contained within the profiles created by RSRPT on I. To prove this containment property, we need the following result.

LEMMA 5.1. *Consider job $SRPT[I^s, i][k+j]$ for $1 \leq j \leq l$. Either this job has 0 remaining processing time in $SRPT(I^s, i+1)$ or SRPT applied at least e_j processing time to this job in interval I_i^s.*

Proofsketch. Consider input instance I. Suppose at release time r_i, we release l additional jobs of size e_1 through e_j. Let this modified input instance be denoted by I' and its stretched variant as $I^{s'}$.

We can reorganize $OptC(I_i')$ to finish these l new jobs instead of processing jobs $OptC(I, i+1)[j]$ for $k+1 \leq j \leq k+l$ in $C(I_i)$. Thus, $OptC(I', i+1)$ will now have $k+l$ complete jobs.

By Corollary 4.2, we know that $SRPT(I^{s'}, i+1) \leq OptC(I', i+1)$ which means that $SRPT(I^{s'}, i+1)$ must have at least $k+l$ complete jobs. Suppose $z \leq l$ of these $k+l$ complete jobs are the newly injected jobs. If so, these z jobs are the z smallest ones with sizes e_l, e_{l-1}, ..., e_{l-z+1}. It then follows that the $k+l-z$ shortest jobs in $SRPT[I^s, i]$ must be complete in $SRPT(I^{s'}, i+1)$.

This implies that these same $k+l-z$ jobs must also be complete in $SRPT(I^s, i+1)$ since removing the extra jobs will not cause these jobs to receive any less processing time in interval I_i. Furthermore, there must be holes of size at least e_{l-z+1} through e_l into which jobs $SRPT[I^s, i][j]$ for $k+l-z+1 \leq j \leq k+l$ can be slotted with the slots being assigned to jobs in inverse order of size; that is, the smallest job gets the largest slot. Thus, the result holds.

COROLLARY 5.1. *$SRPT(I^s, i) \subseteq RSRPT(I, i)$ for $1 \leq i \leq r(I)$.*

Proof. This is by induction on i. The base case with $i = 1$ is trivially true as the profile $RSRPT(I, 1)$ is identical to that of $SRPT(I^s, 1)$ as no jobs have been processed yet.

Let us now assume the result holds for $1 \leq i < r(I)$ and we wish to show it holds for $i+1$. We first observe by Corollary 4.1 that $SRPT(I^s, i) \subseteq RSRPT(I, i)$ implies that $SRPT[I^s, i] \subseteq RSRPT[I, i]$.

Looking at how we construct $RSRPT(I_i)$, the result follows as we always strive to maintain the containment property. The only place where containment might be violated is the assignment of e_j units to jobs $RSRPT[I, i][k+j]$ for $1 \leq j \leq l$. However, Lemma 5.1 shows that $SRPT(I_i^s)$ either finishes job $SRPT[I^s, i][k+j]$ or assigns it at least e_j units of processing.

We now argue on a job by job basis that each job in $SRPT(I^s)$ delays other jobs by less time than they do in $RSRPT(I)$. Thus, the total idle time incurred by SRPT on I^s is no more than the total idle time incurred by RSRPT on I.

THEOREM 5.1. *For any input instance I, the total idle time of $SRPT(I^s)$ is no larger than the total idle time incurred by $RSRPT(I)$.*

Proofsketch. Consider an arbitrary job J_j in I and I^s. Our claim is that the amount of idle time incurred by other jobs that is charged to J_j in $SRPT(I^s)$ by the SRPT-charging scheme is no more than the idle time incurred by other jobs that is charged to the corresponding job in $RSRPT(I)$ (with some possible swapping due to tie-breaking). This follows from the fact that for $1 \leq i \leq r(I)$, $SRPT(I^s, r_i) \subseteq RSRPT(I, i)$. Thus, whenever job J_j is executed in $SRPT(I^s)$, the number of jobs larger than it is at most the number of jobs larger than its corresponding job in $RSRPT(I)$. The result follows.

5.3 Opt incurs at least as much idle time as RSRPT

LEMMA 5.2. *For* $1 \leq i \leq r(I)$, $RSRPT(I,i) \leq OptC(I,i)$.

Proofsketch. We prove this by induction on i. The base case with $i = 1$ is trivially true as the profiles are identical because no jobs have been processed yet.

Let us now assume the result holds for $i < r(I)$. We now need to show the result holds for $i+1$. We first observe that $RSRPT(I,i) \leq OptC(I,i)$ and $OptC(I,i) \leq Opt(I,i)$ imply that $RSRPT(I,i) \leq Opt(I,i)$. We next observe that $RSRPT(I,i) \leq Opt(I,i)$ implies that $RSRPT[I,i] \leq Opt[I,i]$.

We now argue that RSRPT spends no more time on the first k completed jobs in I_i than C does. Suppose this were not true. That would mean that RSRPT spends $Y > 0$ extra time units on the first completed k jobs than C does. This would mean that the prefix sum of the first k jobs in $Opt[I,i]$ must be Y smaller than the prefix sum of the first k jobs in $RSRPT[I,i]$, but this is not possible because $RSRPT[I,i] \leq Opt[I,i]$. This also proves that the value X in the construction of $RSRPT(I_i)$ must be non-negative.

Suppose that $RSRPT(I,i+1)$ is not less than $OptC(I,i+1)$. Then there must be a smallest integer j such that the sum of the first j jobs of $RSRPT(I,i+1)$ is strictly larger than the sum of the first j jobs of $OptC(I,i+1)$. A similar argument shows that no such j exists.

This leads to the following result.

COROLLARY 5.2. *The idle time incurred by partial schedule $RSRPT(I_i)$ must be at most the idle time incurred by partial schedule $C(I_i)$ within $OptC(I_i)$ as determined by SRPT charging.*

This corollary clearly implies that the the total idle time incurred by $RSRPT(I)$ is no larger than than the idle time incurred by $Opt(I)$, and our result is proven.

6 Open problems

We have shown that SRPT optimally uses sufficiently faster machines with respect to minimizing total flow time. Some interesting open problems include the following. Do any non-migratory algorithms also have this property? We have shown that existing non-migratory algorithms are not s-speed $1/s$-competitive algorithms for any $s \geq 1$. Another interesting question is to analyze how well SRPT performs using machines of speed $1 < s < 2 - 1/m$? In particular, can we show that SRPT is a $(1+\epsilon)$-speed $O(1)$-competitive algorithm for minimizing total flow time (as well as other metrics)? Also, what is the smallest s such that SRPT (or any other online algorithm) is an s-speed 1-competitive algorithm? This question addresses the tradeoff between faster machines and lack of knowledge of the future.

References

[1] E. Anderson and C. Potts. On-line scheduling of a single machine to minimize total weighted completion time. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms*, pages 548–557, 2002.

[2] Nir Avrahami and Yossi Azar. Minimizing total flow time and total completion time with immediate dispatching. In *Proceedings of the 15th ACM Symposium on Parallel Algorithms and Architectures*, pages 11–18. ACM, 2003.

[3] Baruch Awerbuch, Yossi Azar, Stefano Leonardi, and Oded Regev. Minimizing the flow time without migration. *SIAM Journal on Computing*, 31:1370–1382, 2001.

[4] Chandra Chekuri, Sanjeev Khanna, and Amit Kumar. Multi-processor scheduling to minimize L_p norms of flow and stretch. Manuscript, 2003.

[5] Chandra Chekuri, Sanjeev Khanna, and An Zhu. Algorithms for weighted flow time. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, pages 84–93. ACM, 2001.

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

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

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

[9] S. Leonardi. A simpler proof of preemptive flow-time approximation. In *Approximation and On-line Algorithms*, Lecture Notes in Computer Science. Springer, 2003.

[10] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 110–119, 1997.

[11] C. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. *Algorithmica*, 32:163–200, 2002.