SRPT Optimally Utilizes Faster Machines to Minimize Flow Time

ERIC TORNG

Michigan State University

AND

JASON MCCULLOUGH

University of Illinois Urbana-Champaign

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.

An extended abstract of this article initially appeared in the *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA04)* (New Orleans, LA, Jan. 11–13), ACM, New York, 2004, pp. 350–358.

E. Torng was supported in part by National Science Foundation (NSF) grants CCR 9701679 and CCR 0105283. J. McCullough was supported in part by NSF grant CCR 9701679, an MSU professional assistantship, and an MSU summer research internship.

This work was performed while J. McCullough was an undergraduate student at Michigan State University.

Authors' addresses: E. Torng, Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48224, e-mail: torng@msu.edu; J. McCullough, Department of Mathematics, 1409 West Green Street, University of Illinois at Urbana-Champaign, Urbana, IL 61801, e-mail: jmccullo@math.uiuc.edu.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*sequencing and scheduling*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: SRPT, scheduling, resource augmentation, flow time, parallel machines

ACM Reference Format:

Torng, E., and McCullough, J. 2008. SRPT optimally utilizes faster machines to minimize flow time. ACM Trans. Algor. 5, 1, Article 1 (November 2008), 25 pages. DOI = 10.1145/1435375.1435376 http://doi.acm.org/10.1145/1435375.1435376

1. *Introduction*

In this article, 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 $\{1, 2, \ldots, n\}$. Each job j has a release time 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.

For any input instance I, a schedule $S(I)$ is an assignment of jobs to machines satisfying the following properties. A job can be assigned to a machine only after its release time r_j. A job can run on only one machine at a time, and a machine can process only one job at a time. We consider a preemptive and migratory scheduling model where a job may be interrupted and subsequently resumed on any machine with no penalty. Assuming machines have speed-1, each job must be assigned to some machine for p_j time units. For any input instance I, any job $j \in I$, any time $t \geq r_j$, and any schedule $S(I)$, let $p(j, t, S(I))$ be job j's remaining length at time t in schedule $S(I)$. Let $C_j(S(I))$ denote the completion time of job j in schedule $S(I)$; that is, the smallest time t such that $p(j, t, S(I)) = 0$. As with most of the following notation, we will omit the schedule $S(I)$ when the schedule is understood from context, particularly in our examples. The *flow time* of job j in schedule $S(I)$ is $F_j(S(I)) = C_j(S(I)) - r_j$. When restricted to speed-1 machines, the *idle time* or *delay* of job j in schedule $S(I)$ is $D_j(S(I)) = F_j(S(I)) - p_j = C_j(S(I)) - r_j - p_j$. The *total flow time* of schedule $S(I)$ is $F(S(I)) = \sum_j F_j(S(I))$. In the classic notation of Graham et al. [1979], 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*, $C(S(I)) = \sum_j C_j(S(I))$, or *total idle time*, $D(S(I)) = \sum_j D_j(S(I))$.

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.

Example 1. Consider an input instance $I = \{(0, 1), (0, 1), (0, 1), (0, 4), (3, 3), (3, 3)\}$ where a job i is specified by the ordered pair (r_i, p_i), and suppose we have two speed-1 machines. SRPT will schedule the two machines as follows. On machine 1, SRPT will execute and complete job 1 in the time interval $[0, 1)$, job 3 in $[1, 2)$, be idle in $[2, 3)$, and execute and complete job 5 in $[3, 6)$. On machine 2, SRPT will execute and complete job 2 in $[0, 1)$, job 4 in $[1, 5)$, and job 6 in $[5, 8)$. At time 3, the remaining length of job 4 is 2, so job 4 is not preempted by either job 5 or job 6 when they are released at time 3. It follows that $D_1 = D_2 = D_5 = 0$, $D_3 = D_4 = 1$, and $D_6 = 2$. Thus, $D(SRPT(I)) = 4$ and $F(SRPT(I)) = 17$.

When the number of machines $m = 1$, SRPT is known to be an optimal algorithm for this problem. One proof of its optimality is given by Schrage [1968]. When $m \geq 2$ but all jobs are released at time 0, SRPT is again optimal [Conway et al. 1967]. Note, this implies SRPT minimizes total completion time, total flow time, and total idle time. When $m \geq 2$ and there are an unbounded number of release times, this problem is known to be NP-complete. Leonardi and Raz [1997] 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. 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 Leonardi [2003]. No offline algorithm with a superior approximation ratio is known for this problem.

Kalyanasundaram and Pruhs [2000] popularized the usage of resource augmentation as a method for analyzing online algorithms, in particular online scheduling algorithms. 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 article, 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. [2002]. 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. [2002]. 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 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. [2002].

THEOREM 1.1. *If A is an s-speed ρ-competitive algorithm for minimizing total flow time in any model (preemptive or non-preemptive, clairvoyant or non-clairvoyant, online or offline), then A is an s-stretch ρs-competitive algorithm for minimizing total flow time in the same model.*

PROOF. For any input instance I, I^s is the identical input instance except job i has release time $r_i s$ for $1 \leq i \leq n$. Thus, at any time ts, the situation faced by 1-speed A on I^s is identical to the situation faced by s-speed A on I; that is, the same jobs with the exact same remaining lengths are available.

Let C_j and F_j denote the completion time and flow time, respectively, of job j when s-speed A schedules input instance I, and C'_j and F'_j denote the completion time and flow time, respectively, of job j when 1-speed A schedules input instance

I^s. We have $C'_j = sC_j$ for $1 \leq j \leq n$. Combining this with the above release time relationship, we see that $F'_j = sF_j$ for $1 \leq j \leq n$, and the result follows. $\square$

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. [2002] where they proved that SRPT is an s-speed 1-competitive algorithm for this problem when $s \geq 2-1/m$. 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. We give a formal proof for the non-migratory algorithms developed by Awerbuch et al. [2001] and Chekuri et al. [2001]. Similar arguments could be applied to the algorithm of Avrahami and Azar [2003]. 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 article 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 [2004] 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 [Kalyanasundaram and Pruhs 2000; Coulston and Berman 1999; Edmonds 2000]. Edmonds [2000] 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 round robin 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. More recently, Chekuri et al. [2004] have shown that the algorithm of Avrahamai and Azar is a $(1+\epsilon)$-speed $O(1/\epsilon)$-competitive algorithm for this problem (and others). 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. Since this work, Bussema and Torng [2006] showed that the shortest job first (SJF) and SRPT are also $(1+\epsilon)$-speed $O(1/\epsilon)$-competitive algorithms for this problem.

The rest of this article 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 of our main result. In Section 4, we introduce some building blocks for the proof such as profiles, partial schedules, and canonical schedules. 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 in Section 6.

2. *Bounds on Nonmigratory Algorithms*

The key idea in algorithms that eliminate migration is the idea of classifying jobs by size [Awerbuch et al. 2001; Chekuri et al. 2001; Avrahami and Azar 2003]. In Awerbuch et al. [2001], 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 Chekuri et al. [2001] and Avrahami and Azar [2003], 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 Chekuri et al. [2001], they optimize their algorithm by identifying the best possible constant c.

The algorithms of Awerbuch et al. [2001] and Chekuri et al. [2001] 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 Awerbuch et al. [2001] and Chekuri et al. [2001] 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 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 Chekuri et al. [2001], 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 formally show that the algorithms of Awerbuch et al. [2001] and Chekuri et al. [2001] cannot be s-speed $1/s$-competitive algorithms for this problem. We use A to denote any implementation of the non-migratory algorithms of Awerbuch et al. [2001] and Chekuri et al. [2001].

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 2.2. *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 $(m/s)(2c - 2\epsilon - \delta + 1)$. 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 also holds for migratory versions of the above algorithms. □

LEMMA 2.3. *A is at best an s-speed $(\frac{c}{c-1} - \frac{m}{c^m-1})\frac{1}{s}$-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 $(1/s)\sum_{k=0}^{m-1}(m - k)c^k + \epsilon(m - 1)/s = (1/s)[(c^{m+1} - c)/(c - 1)^2 - m/(c - 1) + (m - 1)\epsilon]$. 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 be applied to the immediate dispatch algorithm of Avrahami and Azar [2003]. We omit a formal proof but include this informal discussion for those familiar with that algorithm. For small c, there can exist times where m jobs from different classes are released simultaneously and assigned to the same machine when no other jobs are in the system, thus approaching the bound from Lemma 2.3. On the other hand, for larger c, let x be a large even number and assume m is also large and even. We can carefully release m^2x jobs of size $c - \epsilon$ and cm^2x jobs of size 1 such that $m/2$ of the machines have $2mx$ jobs of size $c - \epsilon$ and the other $m/2$ machines have $2cmx$ jobs of size 1. The total flow time for this schedule with speed-s machines ignoring ϵ is then $(1/s)((c^2 + c)m^3x^2 + cm^2x)$ while the optimal total flow time where both types of jobs are evenly divided among all m machines has a total flow time of $(c^2/2 + 3c/2)m^3x^2 + cm^2x$ yielding a ratio that is $1/s + ((1/s)(c^2 - c)m^3x^2)/((c^2 + c + 2)m^3x^2 + 2cm^2x)$ which for large m and x approaches $1/s + (c^2 - c)/(s(c^2 + c + 2))$. Combining the two bounds, we see that the algorithm is bounded away from being $1/s$-competitive given speed-s machines.

3. *Overview of SRPT Upper Bound Proof*

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.

Example 2. Suppose three 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. If no new jobs are released, SRPT will then complete the job of size 2 at time 3, and both schedules will have a flow time of 5.

However, if more jobs are released at time 2, SRPT runs into trouble. For example, suppose three jobs with lengths $1/2$, $1/2$, and 1 arrive at time 2, three jobs with lengths $1/4$, $1/4$, $1/2$ arrive at time 3, three jobs with lengths $1/8$, $1/8$, and $1/4$ arrive at time $7/2$, and so on. Continuing in this fashion, we can create a situation where SRPT has arbitrarily more unfinished jobs than Opt by time 4. If we then feed a long stream of very small jobs starting just after Opt has completed all released jobs (a little bit before time 4) such that any algorithm must finish these newly released jobs before any older jobs, this leads to SRPT having an arbitrarily larger flow time than Opt.

The problem is that SRPT completes less work in interval [0, 2) than Opt does. Intuitively, SRPT has not "kept up with" Opt. We will define a formal notion of "keeping up" in the building blocks section. If we increase the speed s of SRPT's machines, it will still be behind Opt at time 2 until the speed s reaches $3/2 = 2-1/2$, at which point it will complete all three jobs released at time 0 by time 2. For example, suppose three jobs are released at time 0 with lengths 1, 1, and 2, and another three jobs are released at time 2 with lengths 1, 1, and 2. If SRPT has speed $s \geq 3/2$, all the jobs released at time 0 will be completed by time 2, the second release time. SRPT will then process both batches of three jobs s times faster than Opt which leads to SRPT having a flow time s times smaller than Opt. On the other hand, if SRPT has speed $s < 3/2$, the job of size 2 released at time 0 will not be completed by time 2. This will delay one of the jobs of size 1 released at time 2. That is, the two batches of jobs will overlap with the first batch slightly delaying the second batch. Thus, SRPT will not have a flow time s times smaller than Opt.

3.2. PROOF OUTLINE. Our goal is to show that with speed-s machines, where $s \geq 2 - 1/m$, SRPT not only overcomes the issues above, it actually achieves a flow time that is at least s times smaller than that of Opt with speed-1 machines. A key idea 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 need to prove that SRPT is an s-stretch 1-competitive algorithm for $s \geq 2 - 1/m$. Stated another way, we need to prove that $F(\mathit{SRPT}(I^s)) \leq F(\mathit{Opt}(I))$ for any input instance I. This is equivalent to showing $D(\mathit{SRPT}(I^s)) \leq D(\mathit{Opt}(I))$ for any input instance I since any algorithm that minimizes total flow time also minimizes total idle time. We will focus on total idle time for the remainder of this proof.

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. Within each interval, we need to prove two things. The first is that SRPT "keeps up with" Opt in each interval. The second is that the total idle time incurred within each interval I_i^s by SRPT is no more than the total idle time incurred by Opt within each corresponding interval I_i. However, this second goal often is not true because the stretched interval I_i^s is s times longer than interval I_i so SRPT will often incur more total idle time in that interval than Opt does in the corresponding interval I_i.

To overcome this issue, we define a Relaxed SRPT (RSRPT) algorithm that works on input instance I, not I^s. RSRPT is used only for analysis purposes. We will show that RSRPT "keeps up with" Opt at the end of each interval I_i and that the idle time incurred by RSRPT on each interval I_i is at most the idle time incurred by Opt on I_i. RSRPT is not a real algorithm because it might use illegal schedules where some machines 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 account for this, we will define a method for computing total idle time within an interval that works even with illegal schedules; we call this method *SRPT charging*.

Finally, we need to show that the total idle time incurred by RSRPT for I is at least the total idle 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 that the idle time incurred in *RSRPT*(I) is at least as large as the idle time incurred in *SRPT*(I^s). 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 ith release time in I for $1 \le i \le r(I)$. Without loss of generality, assume $r_1(I) = 0$ for all instances 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 $1 \le i \le r(I) - 1$. Time interval $I_{r(I)}$ is defined to be $[r_{r(I)}, \infty)$. We use I_{i-} to denote the time interval $[0, r_i(I))$ for $1 \le i \le r(I)$. For $1 \le i \le r(I) - 1$, we use $|I_i|$ to denote the length of interval I_i. For any schedule $S(I)$ and $1 \le i \le r(I)$, 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. We will use $D_j(S(I_i))$ to denote the delay experienced by job j in partial schedule $S(I_i)$ in the time interval I_i and $D(S(I_i))$ to denote the total delay incurred by partial schedule $S(I_i)$ in the time interval I_i.

Example 3. Let $I = \{(0, 1), (0, 1), (0, 2), (2, 3), (2, 3)\}$, and suppose there are two speed-1 machines. 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 jobs 1 and 2 in interval $[0, 1)$ and job 3 on machine 1 in interval $[1, 2)$, and $D(\mathit{SRPT}(I_1)) = 1$. The partial schedule *SRPT*(I_2) is to run job 3 on machine 1 in $[2, 3)$, job 4 on machine 2 in $[2, 5)$, and job 5 on machine 1 in $[3, 6)$, and $D(\mathit{SRPT}(I_2)) = 1$.

4.2. PROFILES.

To compare how much work is left in different schedules at various times t where t is typically a release time, we introduce profiles and methods for comparing profiles.

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 time t, denoted $S(I, t)$, to be the nondecreasing vector of remaining lengths of all jobs that were released up to but not including time t. We use $S[I, t]$ to denote the profile that includes jobs released at time t. We define $|S(I, t)|$ and $|S[I, t]|$ to be the number of elements in profiles $S(I, t)$ and $S[I, t]$, respectively.

Definition 4.2. We define $S(I, t)[j]$ and $S[I, t][j]$ to be the jobs with the jth smallest remaining length in profiles $S(I, t)$ and $S[I, t]$, respectively. If two jobs tie for the jth smallest remaining length, $S(I, t)[j]$ (or $S[I, t][j]$) is the one that received more processing time since the previous release time of I. If there is still a tie in amount of processing time received since the closest release time, ties are broken arbitrarily. We overload notation and also use $S(I, t)[j]$ and $S[I, t][j]$ to denote that job's remaining length at time t. Finally, we define $S^j(I, t) = \sum_{q=1}^{j} S(I, t)[q]$ and $S^j[I, t] = \sum_{q=1}^{j} S[I, t][q]$ to be the sum of the j smallest remaining lengths in profiles $S(I, t)$ and $S[I, t]$, respectively.

For most of this article, we are primarily concerned with profiles at release times. To denote the profile at release time r_i for $1 \leq i \leq r(I)$, we will typically use $S(I, i)$ or $S[I, i]$. When we do not use release times, we will use variable letter t to denote the time.

When working with general profiles independent of a specific schedule or input instance, we will use the notation P or P_i in place of $S(I, i)$ or $S[I, i]$. For example, P_i^j denotes the sum of the j smallest remaining lengths in profile P_i.

Definition 4.3. We can compare two profiles P_1 and P_2 if $|P_1| = |P_2|$; that is, they have the same number of elements. We say that a profile P_1 is *smaller than* another profile P_2, denoted $P_1 \leq P_2$, if the following condition holds:

—For $1 \leq i \leq |P_1|$, $P_1^i \leq P_2^i$; that is, 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.

Definition 4.4. We say that a profile P_1 is *contained* by another profile P_2, denoted $P_1 \subseteq P_2$, 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.

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.

Example 4. Let $I = \{(0, 1), (0, 1), (0, 2), (2, 1)\}$ and suppose there is one speed-1 machine. Suppose schedule $S(I)$ schedules job 3 in the interval [0, 2). SRPT will schedule job 1 in the interval [0, 1) and job 2 in the interval [1, 2), so $SRPT(I, 2) = \langle 0, 0, 2\rangle \leq S(I, 2) = \langle 0, 1, 1\rangle$. Furthermore, $SRPT(I, 2)[2] = 0$ while $S(I, 2)[2] = 1$. However, it is not true that $SRPT(I, 2) \subseteq S(I, 2)$.

Occasionally, we will 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. For convenience, we may assume that the jobs of size 0 do not exist and remove them from the instance. Typically this is not necessary.

We now list a few simple observations about profiles.

FACT 4.5. *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. By the definition of $P_1 \leq P_2$, it follows that $|P_1| = |P_2| = n$ for some non-negative integer n. The definition of $P_1 \leq P_2$ also implies that $P_1^n \leq P_2^n$ which is the desired result. □

FACT 4.6. *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).*

PROOF. Let $n = |P_1|$. Let $i+1$ be the index where the job of size x is located in profile P_1' and $j+1$ be the index where the job of size x is located in profile P_2'.

We first consider the case where $P_1 \leq P_2$. For $1 \leq k \leq \min(i, j)$, $P_1'^k = P_1^k \leq P_2^k = P_2'^k$ where the middle inequality holds because $P_1 \leq P_2$. For $\max(i+1, j+1) \leq k \leq n+1$, we have that $P_1'^k = P_1^{k-1} + x$ and $P_2'^k = P_2^{k-1} + x$ and $P_1^{k-1} \leq P_2^{k-1}$. Thus, it follows again that $P_1'^k \leq P_2'^k$. We now consider the intermediate range. Suppose $i \leq j$. For $i+1 \leq k \leq j$, $P_1'^k \leq P_1^k$ because $P_1'^k$ replaces $P_1[k]$ with x in the sum and x is smaller than $P_1[k]$ since x has been slotted before $P_1[k]$ in P_1'. At the same time, we have $P_2'^k = P_2^k$ and $P_1^k \leq P_2^k$. Thus, $P_1'^k \leq P_2'^k$. On the other hand, suppose $i > j$. We have that $P_1'^j \leq P_2'^j$. For $j+1 \leq k \leq i$, $P_1[k] \leq x$ or else x would not be $P_1'[i+1]$. For $j+1 \leq k \leq i$, $P_2[k] \geq x$ or else x would not be $P_2'[j+1]$. Thus, for $j+1 \leq k \leq i$, $P_1'^k = (P_1'^j$ plus $k-j$ terms of size at most $x)$ while $P_2'^k = (P_2'^j$ plus $k-j$ terms of size at least $x)$. This shows that for $j+1 \leq k \leq i$, $P_1'^k \leq P_2'^k$.

We now consider the case where $P_1 \subseteq P_2$. It must be the case that $j \leq i$. For $1 \leq k \leq j$, $P_1'[k] = P_1[k] \leq P_2[k] = P_2'[k]$ where the middle inequality holds because $P_1 \subseteq P_2$. Likewise, for $i+2 \leq k \leq n+1$, $P_1'[k] = P_1[k-1] \leq P_2[k-1] = P_2'[k]$. For $j+1 \leq k \leq i+1$, $P_1'[k] \leq x$ while $P_2'[k] \geq x$ and thus $P_1'[k] \leq P_2'[k]$. □

COROLLARY 4.7. *Let I be any input instance. For any $1 \leq i \leq r(I)$ and $s_1, s_2 \geq 1$, and any schedules $S_1(I)$ and $S_2(I)$ 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), we have $S_1[I^{s_1}, i] \leq S_2[I^{s_2}, i]$ ($S_1[I^{s_1}, i] \subseteq S_2[I^{s_2}, i]$, respectively).*

PROOF. This follows from the previous fact by adding the jobs released at the ith release time one by one to the profiles $S_1(I^{s_1}, i)$ and $S_2(I^{s_2}, i)$. □

Continuing Example 4, this implies that $SRPT[I, 2] = \langle 0, 0, 1, 2\rangle \leq S[I, 2] = \langle 0, 1, 1, 1\rangle$.

Finally, we will show that if a new job j is added to an input instance I, SRPT's profile on the new instance $I \cup \{j\}$ is better than SRPT's profile on the original instance I if we ignore the last element of SRPT's profile at any time.

Definition 4.8. For any input instance I, let $SRPT^{-1}[I, t]$ and $SRPT^{-1}(I, t)$ denote profiles $SRPT[I, t]$ and $SRPT(I, t)$ without their maximum element.

FACT 4.9. *Let I be any input instance, and let j be any job not in I with release time r_j. Then for $t \geq r_j$, $SRPT^{-1}[I \cup \{j\}, t] \subseteq SRPT[I, t]$.*

PROOF. Consider time r_j. It is obvious that $SRPT^{-1}[I \cup \{j\}, r_j] \subseteq SRPT[i, r_j]$ as the addition of job j can only reduce the size of the kth largest element of $SRPT^{-1}[I \cup \{j\}, r_j]$ relative to the kth largest element of $SRPT[I, r_j]$ for $1 \leq k \leq |SRPT[I, r_j]$. Since $SRPT$ always works on the shortest m available jobs, this relationship will remain true until the next job j' in I is released. This means $SRPT^{-1}(I \cup \{j\}, r_{j'}) \subseteq SRPT(I, r_{j'})$. Let $P = SRPT^{-1}(I \cup \{j\}, r_{j'})$, and let P' be the profile that results when job j' is added. Fact 4.6 then implies that $P' \subseteq SRPT[I, r_{j'}]$. We then observe that P' can only differ from $SRPT^{-1}[I \cup \{j\}, r_{j'}]$ in their maximal elements. Furthermore, this difference can only take place if j' is the maximum element of $SRPT[I \cup \{j\}, r_{j'}]$ in which case the maximum element of P' is j' while the maximum element of $SRPT^{-1}[I \cup \{j\}, r_{j'}]$ is the maximum element of $SRPT(I \cup \{j\}, r_{j'})$ which is less than the size of j'. Thus, we see that $SRPT^{-1}[I \cup \{j\}, r_{j'}] \subseteq SRPT[I, r_{j'}]$, and the same logic implies this relationship will hold for all $t \geq r_{j'}$, and the result follows. $\square$

4.3. BUILDING BLOCKS FROM PREVIOUS WORK.

There are two critical ideas we need to borrow from the paper of Phillips et al. [2002] 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 article. 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 4.10. *Consider any input instance I, any legal speed-1 schedule $S(I)$, and any $1 \leq i \leq r(I)$. If $s \geq 2 - 1/m$, then $SRPT(I^s, i) \leq S(I, i)$.*

PROOF. Suppose this result is not true. There must be some input instance I, some release time i, some schedule $S(I)$, and some $k \leq |S(I, i)|$ such that $SRPT^k(I^s, i) > S^k(I, i)$. Let $t = r_i$. Define input instance I_1 to be only the jobs that eventually form the k smallest elements of profile $S(I, i)$. By the result from Phillips et al., it follows that $SRPT^k(I_1^s, t) \leq S^k(I_1, t)$. We then insert jobs from I and I^s back into I_1 and I_1^s one at a time. Applying Fact 4.9, the size of each of the first k elements in each resulting profile at time t does not increase. This implies that

$SRPT^k(I^s, t) \leq SRPT^k(I_1^s, t)$. With transitivity, we get that $SRPT^k(I^s, t) \leq S^k(I, t)$ which is a contradiction, and the result follows. □

4.4. CANONICAL SCHEDULES. For any input instance I, we now define the notion of a canonical partial schedule $S(I_i)$ for $1 \leq i \leq r(I) - 1$.

Definition 4.11. For any input instance I, any legal schedule $S(I)$, and any $1 \leq i \leq r(I) - 1$, we say that the ordered pair of jobs (j, k) is *inverted in* $S(I_i)$ if

—$0 < p(j, r_i, S(I)) < p(k, r_i, S(I))$
—$p(j, r_{i+1}, S(I)) > 0$
—j is processed for less time than job k in partial schedule $S(I_i)$.

Definition 4.12. For any input instance I, any legal schedule $S(I)$, and any $1 \leq i \leq r(I) - 1$, partial schedule $S(I_i)$ is *compliant* if there are no ordered pairs of jobs that are inverted in $S(I_i)$.

Definition 4.13. For any input instance I, any legal schedule $S(I)$, any $1 \leq i \leq r(I) - 1$, and any pair of jobs j and k such that $p(j, r_{i+1}, S(I_i)) > 0$, we define the operation *swap*(j, k) in partial schedule $S(I_i)$ as follows. Find all intervals in $S(I_i)$ where job k is run and job j is not. It is possible that no such intervals exist. In all such intervals, run j instead of k until there are no more such intervals or job j completes. Finally, if there are times where job k is being processed and j is not prior to times where job j is being processed and k is not, swap the jobs until this is no longer true.

LEMMA 4.14. *Consider any input instance I, any legal schedule $S_1(I)$, and any $1 \leq i \leq r(I) - 1$ where partial schedule $S_1(I_i)$ contains two jobs j and k such that $p(j, r_i, S_1(I)) \leq p(k, r_i, S_1(I))$ and $p(j, r_{i+1}, S_1(I)) > 0$. Let $S_2(I)$ denote the schedule up to time r_{i+1} that results from applying operation $swap(j, k)$ in partial schedule $S(I_i)$. It follows that $D(S_2(I_i)) \leq D(S_1(I_i))$ and $S_2(I, i+1) \leq S_1(I, i+1)$.*

PROOF. We are given that $p(j, r_{i+1}, S_1(I)) > 0$, so $swap(j, k)$ is well defined. We observe that $swap(j, k)$ effects no jobs other than jobs j and k; that is, for any job $l \not\in \{j, k\}$, $p(l, r_{i+1}, S_1(I)) = p(l, r_{i+1}, S_2(I))$ and $D_j(S_1(I_i)) = D_j(S_2(I_i))$. Let x denote the total amount of time spent processing jobs j and k in $S_1(I_i)$. Since no other jobs are affected, x is also the total amount of time spent processing jobs j and k in $S_2(I_i)$.

We now consider two possibilities for $p(k, r_{i+1}, S_1(I))$. The first is that $p(k, r_{i+1}, S_1(I)) = 0$ which means job k was completed by time r_{i+1} in schedule $S_1(I)$. We first show that this implies $D(S_2(I_i)) \leq D(S_1(I_i))$. Since job j does not complete in $S_1(I_i)$, it follows that $D_j(S_1(I_i)) + D_k(S_1(I_i)) = |I_i| + C_k(S_1(I_i)) - r_i - x$. Since $0 < p(j, r_i, S_1(I)) \leq p(k, r_i, S_1(I))$, the amount of time that k is running when j is not running must be sufficient to complete job j. Thus, after $swap(j, k)$, job j will be completed in partial schedule $S_2(I_i)$ and $C_j(S_2(I_i)) \leq C_k(S_1(I_i))$. This implies $D_j(S_2(I_i)) + D_k(S_2(I_i)) = |I_i| + C_j(S_2(I_i)) - r_i - x$. It follows that $D_j(S_2(I_i)) + D_k(S_2(I_i)) \leq D_j(S_1(I_i)) + D_k(S_1(I_i))$, and thus $D(S_2(I_i)) \leq D(S_1(I_i))$.

We now show that $S_2(I, i + 1) \leq S_1(I, i + 1)$ when $p(k, r_{i+1}, S_1(I)) = 0$. As noted earlier, for any job $l \not\in \{j, k\}$, $p(l, r_{i+1}, S_1(I)) = p(l, r_{i+1}, S_2(I))$. We have shown $p(j, r_{i+1}, S_2(I)) = 0 = p(k, r_{i+1}, S_1(I))$. Finally, since the extra time given to job j in $S_2(I_i)$ comes from job k and not other jobs, it follows that

$p(k, r_{i+1}, S_2(I)) = p(j, r_{i+1}, S_1(I))$. Thus, $S_1(I, i+1) = S_2(I, i+1)$ with jobs j and k trading places in the two profiles.

The second possibility is that $p(k, r_{i+1}, S_1(I)) > 0$ which means job k was not completed by time r_{i+1} in schedule $S_1(I)$. We first show that this implies $D(S_2(I_i)) \leq D(S_1(I_i))$. Since neither job completes in $S_1(I_i)$, $D_j(S_1(I_i)) + D_k(S_1(I_i)) = 2|I_i| - x$. Since the total amount of processing both jobs receive in $S_2(I_i)$ is still x, it follows that $D_j(S_2(I_i)) + D_k(S_2(I_i)) \leq 2|I_i| - x$; the delay of these two jobs may fall if job j completes in $S_2(I_i)$. Thus, $D(S_2(I_i)) \leq D(S_1(I_i))$.

We now show that $S_2(I, i+1) \leq S_1(I, i+1)$ when $p(k, r_{i+1}, S_1(I)) > 0$. Again, as noted earlier, for any job $l \notin \{j, k\}$, $p(l, r_{i+1}, S_1(I)) = p(l, r_{i+1}, S_2(I))$. We now consider what happens as we transfer processing time from job k to job j. Either job j will be completed or all the time intervals when job k was being processed but job j was not in $S_1(I_i)$ will now be time intervals when job j is being processed but job k is not in $S_2(I_i)$. Since $p(j, r_i, S_1(I)) \leq p(k, r_i, S_1(I))$, it must be the case that $p(j, r_{i+1}, S_2(I)) \leq p(k, r_{i+1}, S_1(I))$. Let $y = p(k, r_{i+1}, S_1(I)) - p(j, r_{i+1}, S_2(I))$. The above implies $y \geq 0$. It follows that $y = p(k, r_{i+1}, S_2(I)) - p(j, r_{i+1}, S_1(I))$. Thus $S_2(I, i+1)$ is basically the same as $S_1(I, i+1)$ with jobs j and k trading places in the two profiles except that job j in $S_2(I, i+1)$ is smaller than job k in $S_1(I, i+1)$ by y and job k in $S_2(I, i+1)$ is larger than job j in $S_1(I, i+1)$ by y. Since any prefix sum which includes job k in $S_2(I, i+1)$ will also include job j in $S_2(I, i+1)$, it follows that for $1 \leq l \leq |S_1(I, i+1)|$, $S_2^l(I, i+1) \leq S_1^l(I, i+1)$. □

Definition 4.15. For any input instance I, any legal schedule $S(I)$, and any $1 \leq i \leq r(I) - 1$, partial schedule $S(I_i)$ is *canonical* if it has the following properties:

(1) $S(I_i)$ is compliant.
(2) Each machine processes at most one job that is not completed in $S(I_i)$.
(3) On any machine, no unfinished job is processed prior to any finished job in $S(I_i)$.

It is important to observe we only define canonical partial schedules up to the final release time $r(I)$ and not after $r(I)$.

LEMMA 4.16. *For any input instance I, any legal schedule $S_1(I)$, and any $1 \leq i \leq r(I) - 1$, there exists a schedule $S_2(I)$ such that $S_2(I_{i-})$ is identical to $S_1(I_{i-})$, $S_2(I_i)$ is canonical, $D(S_2(I_i)) \leq D(S_1(I_i))$, and $S_2(I, i+1) \leq S_1(I, i+1)$.*

PROOF. We will specify schedule $S_2(I)$ up to time r_{i+1} because what happens in $S_2(I)$ after time r_{i+1} is irrelevant. We first make $S_2(I_{i-}) = S_1(I_{i-})$. To describe $S_2(I_i)$, we need the following definition. Any job that is processed in a schedule $S(I_i)$ but not finished by time r_{i+1} is defined to be an incomplete job. We will create schedule $S_2(I_i)$ from $S_1(I_i)$ so that $S_2(I_i)$ is compliant, each machine processes at most one incomplete job in $S_2(I_i)$, and the incomplete jobs will satisfy the following containment property. Suppose we number the incomplete jobs so that $j < k$ implies that $p(j, r_{i+1}, S_2(I_i)) \leq p(k, r_{i+1}, S_2(I_i))$. The containment property of $S_2(I_i)$ is that whenever incomplete job k is running, incomplete job j is running as well. For example, any time any incomplete job is running in $S_2(I_i)$, incomplete job 1 must be running in $S_2(I_i)$. If incomplete job 4 is running in $S_2(I_i)$, then incomplete jobs 1, 2, and 3 must also be running in $S_2(I_i)$.

We next number the jobs in $S_2[I, i]$ based on their remaining processing times at release time r_i in schedule $S_1(I)$. That is, if $j < k$, then $p(j, r_i, S_1(I_i)) \leq p(k, r_i, S_1(I_i))$. We break ties based on their remaining processing times at release time r_{i+1} in $S_1(I)$. that is, if $j < k$ and $p(j, r_i, S_1(I_i)) = p(k, r_i, S_1(I_i))$, then $p(j, r_{i+1}, S_1(I_i)) \leq p(k, r_{i+1}, S_1(I_i))$. Remaining ties are broken arbitrarily. For a numbered incomplete job i, we use $N(i)$ to denote its number in this total ordering.

We create $S_2(I_i)$ with these properties using the following process. Initially define $S_2(I_i) = S_1(I_i)$. Let $j*$ be the lowest numbered job that is unfinished at time r_{i+1}. If no higher numbered jobs received any processing in $S_2(I_i)$, then we are done as the schedule must be compliant and has no incomplete jobs. Otherwise, update schedule $S_2(I_i)$ by performing $swap(j*, k)$ where k is a higher numbered job that received some processing in $S_2(I_i)$ until either all swaps have taken place or job $j*$ is completed. By Lemma 4.14, each swap operation reduces $D(S_2(I_i))$ and profile $S_2(I, i+1)$. If $j*$ is completed, then return to the top of the loop.

If $j*$ is not completed, then $j*$ is incomplete job 1. Perform swaps to move all executions of incomplete job 1 to machine 1. Because we have swapped with all other incomplete jobs, there is no time when job 1 is not running and some other incomplete job is running. Thus, the containment property holds for incomplete job 1.

Set $k = 2$. While $k \leq m$ and there are incomplete jobs that have not been numbered, let $j*$ be the lowest numbered job greater than $N(k-1)$ that is unfinished at time r_{i+1}. Job $j*$ will be incomplete job k. For any job $j > j*$, update $S_2(I_i)$ by performing $swap(j*, j)$. Again, by Lemma 4.14, each swap operation reduces $D(S_2(I_i))$ and profile $S_2(I, i+1)$. Note job $j*$ cannot complete because any time it is running must be contained within the time incomplete job 1 is running, and $p(N(1), r_i, S_2(I)) \leq p(j*, r_i, S_2(I))$. Furthermore, there is no time when an incomplete job is running on machines k through m and job $j*$ is not. Thus, the containment property holds for job $j*$. Swap all executions of job $j*$ to machine k, increment k by 1, and return to the beginning of this loop. When we exit this loop, the schedule $S_2(I_i)$ is compliant.

Finally, we manipulate $S_2(I_i)$ so that each machine processes the completed jobs in $S_2(I_i)$ before that machine does any processing of the at most one incomplete job it executes in $S_2(I_i)$. Break interval I_i into maximal subintervals $I(1), I(2), \ldots,$ such that during subinterval $I(j)$, the jobs being processed by the m machines are unchanging. We call an interval $I(j)$ a q-interval if it contains exactly q jobs that are not completed in $S_2(I_i)$.

We now show that if $I(i)$ is a j-interval while $I(i+1)$ is a q-interval where $j > q$, then we can swap $I(i)$ and $I(i+1)$ without increasing idle time. Assume for the moment that $I(i)$ and $I(i+1)$ have the same length. We now need to identify jobs that can swap so that $I(i+1)$ is now a j-interval, $I(i)$ is a q-interval, we do not increase idle times, and we do not increase the final profile. By our containment property, we know that the q incomplete jobs in $I(i+1)$ are incomplete job 1 through incomplete job q and the j incomplete jobs in $I(i)$ are incomplete job 1 through incomplete job j. Thus, the extra incomplete jobs in $I(i)$ are incomplete jobs $q+1$ through incomplete job j. Furthermore, there are $j-q$ extra completed jobs in interval $I(i+1)$ that are not in $I(i)$. So, the first thing we do is update $S_2(I_i)$ by rearranging the completed jobs in interval $I(i+1)$ so that the $j-q$ completed jobs on machines $q+1$ through j are different from any of the completed jobs in interval $I(i)$. This has no effect on $D(S_2(I_i))$ or $S_2(I, i+1)$ since jobs have only been migrated. We now update $S_2(I_i)$ by swapping the incomplete jobs in $I(i)$ on machines $q+1$ through j with the completed jobs in $I(i+1)$ on machines $q+1$ though j. This swap results in a legal schedule because of our precautions to ensure that the jobs moved were not already run in the subinterval they were moving to. Profile $S_2(I_i)$ is unaffected because the only changes made were when jobs were scheduled, not how much time each job received. $D(S_2(I_i))$ can only decrease since the only jobs moved later are incomplete jobs which means their overall idle time is unaffected. Finally, the containment property for incomplete jobs is not affected by this operation.

Now suppose $I(i)$ is longer than $I(i+1)$. In this case, we divide interval $I(i)$ into an initial piece $I(i)'$ that has length identical to $I(i+1)$ and a second piece that has the remainder of $I(i)$. We now update $S_2(I_i)$ by performing the swap between $I(i)'$ and $I(i+1)$ as described above. Likewise, if $I(i+1)$ is longer than $I(i)$, we divide $I(i+1)$ into a second piece $I(i+1)'$ that has length identical to $I(i)$ and an initial piece that has the remainder of $I(i+1)$. We then update $S_2(I_i)$ by performing the swap between $I(i)$ and $I(i+1)'$ as described above. In either case, at the end of the swap, we have three subintervals, but all the desired properties hold for the three subintervals.

We continue updating $S_2(I_i)$ by performing these swaps until there are no swaps left to perform. Schedule $S_2(I_i)$ is still compliant, each machine has at most one incomplete job, and these jobs are processed after any completed jobs on the same machine. Furthermore, $D(S_2(I_i)) \leq D(S_1(I_i))$ and $S_2(I, i+1) \leq S_1(I, i+1)$, and the result follows. $\square$

A crucial property of canonical partial schedules is that they do not affect the relative position of jobs ordered by remaining processing time.

COROLLARY 4.17. *Consider any input instance I, any i such that $1 \leq i < r(I)$, and any schedule $S(I)$ where $S(I_i)$ is a canonical partial schedule. Consider any pair of jobs j_1 and j_2 in profile $S(I,i)$ where $p(j_1, r_i) < p(j_2, r_i)$. Then $p(j_1, r_{i+1}) \leq p(j_2, r_{i+1})$.*

PROOF. This follows from the fact that canonical partial schedules are compliant. $\square$

4.5. OPTC. Our primary use for canonical schedules will be the following. Given any input instance I, and any optimal schedule $Opt(I)$, for each interval I_i for $1 \leq i < r(I)$, we associate a canonical partial schedule $OptC(I_i)$ whose existence is guaranteed by Lemma 4.16. Note there may be more than one such canonical partial schedule $OptC(I_i)$ possible. We can use any such schedule as $OptC(I_i)$. Finally, we define $OptC(I_{r(I)}) = Opt(I_{r(I)})$.

Note that the partial schedules $OptC(I_i)$ for $1 \leq i \leq r(I)$ cannot be concatenated to form a complete schedule for I_i. That is, when forming $OptC(I_2)$, we assume that $Opt(I)$ was followed exactly in the time interval $[0, r_2)$; in particular, $Opt(I_1)$ was used for I_1 instead of $OptC(I_1)$.

We will use these canonical schedules as follows. First, because of Lemma 4.16, we know that $D(OptC(I_i)) \leq D(Opt(I_i))$. Thus, we will use $D(OptC(I_i))$ as a lower bound on $D(Opt(I_i))$. Second, also because of Lemma 4.16, we know that if we replace $Opt(I_i)$ with $OptC(I_i)$, we obtain a profile P where $P \leq Opt(I, i+1)$. We use the notation $OptC(I, i+1)$ to define this profile P. That is, for $1 \leq i \leq r(I)-1$, we define profile $OptC(I, i+1)$ to be the profile that results from concatenating schedule $Opt(I_{i-1})$ with schedule $OptC(I_i)$. We define $OptC(I, 1)$ to be the empty profile containing no jobs.

Example 5. Let $I = \{(0, 1), (0, 1), (0, 8), (1, 1), (1, 1), (2, 1), (2, 1), (3, 1), (3, 1)\} \cup \{(x, 1), (x, 1) \mid 8 \leq x \leq 999\}$, and suppose we have 2 machines. The optimal schedule is to allocate machine one to the job of size 8 in the interval $[0, 8)$ and to jobs of size 1 in the time interval $[8, 1000)$ while allocating machine two to jobs of size 1 in the time interval $[0, 1000)$. $Opt(I_1)$ thus is to allocate one machine to the job of size 8 in the time interval $[0, 1)$ and one machine to a job of size 1 in the time interval $[0, 1)$. On the other hand, the only canonical partial schedule $OptC(I_1)$ is to allocate both machines to the two available jobs of size 1. Thus, $Opt(I, 2) = \langle 0, 1, 7\rangle$ while $OptC(I, 2) = \langle 0, 0, 8\rangle$. Similarly, $Opt(I_2)$ also allocates one machine to the job with remaining length 7 and one machine to a job of size 1 while the only canonical partial schedule $OptC(I_2)$ allocates both machines to jobs of size 1. In this case, $Opt(I, 3) = \langle 0, 0, 1, 1, 6\rangle$ while $OptC(I, 3) = \langle 0, 0, 0, 1, 7\rangle$.

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. We will prove that $D(SRPT(I^s)) \leq D(Opt(I))$ where $s \geq 2 - 1/m$.

By definition, $D(Opt(I)) = \sum_{i=1}^{r(I)} D(Opt(I_i))$. By Lemma 4.16, $D(OptC(I_i)) \leq D(Opt(I_i))$ for $1 \leq i \leq r(I)$. Ideally, we would show that for each i in the range $1 \leq i \leq r(I)$ that $D(SRPT(I_i^s)) \leq D(OptC(I_i))$. However, this may not be true as I_i^s is s times longer than I_i for $1 \leq i < r(I)$.

In order to make such a comparison, we define for analysis purposes only the Relaxed Shortest Remaining Processing Time (RSRPT) schedule that will work on input instance I, not I^s. We will show that for $1 \leq i < r(I)$, $D(RSRPT(I_i)) \leq D(OptC(I_i))$ and that $RSRPT(I, i) \leq OptC(I, i)$ for $1 \leq i \leq r(I)$. On the other hand, we will also show that $SRPT(I_i^s)$ is contained by $RSRPT(I_i)$ for $1 \leq i \leq r(I)$. This implies that the total idle time incurred by $RSRPT(I)$ is at least the total idle time incurred by $SRPT(I^s)$.

5.1. Relaxed SRPT (RSRPT).

We construct $RSRPT(I)$ interval by interval. For $1 \leq i \leq r(I)$, we create $RSRPT(I_i)$ by combining elements of $OptC(I_i)$ with constraints imposed by $SRPT(I^s, i + 1)$. Some of the partial schedules $RSRPT(I_i)$ may be illegal, but we still concatenate them together to form $RSRPT(I)$. 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 omit s to simplify notation.

Before we construct $RSRPT(I_i)$, we define the following quantities based on $OptC(I_i)$. For $0 \leq i \leq r(I) - 1$, let $OptC(I, i + 1)$ denote the profile at the end of $OptC(I_i)$ not including jobs released at time r_{i+1}. Remember $OptC(I, 1)$ is an empty profile containing no jobs. Let k be the number of jobs with zero remaining length in $OptC(I, i + 1)$ (some of these jobs may have been completed prior to r_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 $OptC(I_i)$ 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 $OptC(I_i)$ 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 ensures 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 $a_y$ processing time to $RSRPT[I, i][y]$ until
        $RSRPT[I, i][y]$ is complete OR
        $RSRPT(I, i + 1)[y] = SRPT(I^s, i + 1)[y]$ OR
        We have added $X_y$ processing time.
    Let $X_{y+1} = X_y - a_y$.
    Increment $y$ by 1
}
```

Example 6. Let $I = \{(0, 4), (0, 4), (0, 32), (4, 4), (4, 4), (4, 4), (4, 4), (10, 4), (10, 4)\} \cup \{(x, 1), (x, 1) \mid 32 \leq x \leq 1000\}$, and suppose there are 2 machines and $s = 3/2$. For I^s, the first four release times are 0, 6, 15, and 48, and the last release time is 1500.

$Opt(I_1)$ is the following: schedule the large job of size 32 on machine 1 in the interval [0, 4) while using the other machine to complete a size 4 job. There is only one possibility for $OptC(I_1)$ which is to use both machines to complete size 4 jobs. Meanwhile, $SRPT(I_1^s)$ completes both jobs of size 4 on the two machines in interval [0, 4) and then schedules the job of size 32 on machine 1 in the interval [4, 6) while machine 2 is idle in interval [4, 6). Finally, $RSRPT(I_1) = OptC(I_1)$.

Things get more interesting when we consider I_2. In this case, $Opt(I_2)$ is to schedule the large job on machine 1 in the interval [4, 10) while completing one size 4 job on machine 2 in [4, 8) and executing another size 4 job on machine 2 in interval [8, 10). There are several possibilities for $OptC(I_2)$ including (a) completing 3 size 4 jobs by time 10 by scheduling one on machine 1 in [4, 8), one on machine 2 in [6, 10), and the third on machine 1 in [8, 10) and machine 2 in [4, 6). Another option is (b) completing two size 4 jobs by time 8 and finishing half of two more size 4 jobs by time 10. Suppose we go with option (b). In this case, we have $OptC(I, 3) = \langle 0, 0, 0, 2, 2, 4, 28\rangle$, so $k = 3$, $l = 2$, $e_1 = 2$, and $e_2 = 2$. Meanwhile $SRPT(I_2^s)$ completes two jobs of size 4 in the interval [6, 10), two jobs of size 4 in the interval [10, 14), and, in the interval [14, 15), schedules the job with remaining length 30 on machine 1 while idling machine 2. Thus, $SRPT(I^s, 3) = \langle 0, 0, 0, 0, 0, 0, 29 \rangle$. So, when we compute $RSRPT(I_2)$, we first ensure that at least three jobs are complete by time 10. Note that two jobs of size 4 were completed in $RSRPT(I_1)$, so this means RSRPT must complete at least one more size 4 job. So, on machine 1, RSRPT completes one size 4 job in the interval [4, 8). Next, we take the next two shortest jobs in $RSRPT[I, 2]$ which are two more size 4 jobs and assign to them 2 units of work each as $e_1 = 2$ and $e_2 = 2$. In this case, that means we schedule one job on machine 2 in the interval [4, 6) and the other job on machine 1 in the interval [8, 10). This leaves us with $X = 12 - 8 = 4$. We now distribute the $X_4 = X = 4$ remaining time units to the remaining jobs. We first consider the 4th job in $RSRPT[I, 2]$, namely the size 4 job scheduled on machine 2 in the interval [4, 6). The 4th job in $SRPT(I^s, 3)$ has remaining length 0, so it is allowable to devote 2 remaining time units to complete this job in $RSRPT(I_2)$, so $RSRPT(I_2)$ completes this job on machine 2 in the interval [4, 8). The quantity $X_5 = X_4 - 2 = 2$. We next consider the 5th job in $RSRPT[I, 2]$, namely the size 4 job scheduled on machine 1 in the interval [8, 10). The 5th job in $SRPT(I^s, 3)$ has remaining length 0, so it is allowable to devote the $X_5 = 2$ remaining time units to complete this job in $RSRPT(I_2)$, so $RSRPT(I_2)$ completes this job on machine 1 in the interval [8, 12). The quantity $X_6 = X_5 - 2 = 0$ and we are done. Note that $RSRPT(I_2)$ is an illegal schedule as it extends into the time interval [10, 12).

5.2. SRPT Charging.

For any input instance I, we define a method we call *SRPT Charging* for computing $D(RSRPT(I_i))$ for $1 \leq i \leq r(I)$. Furthermore, we will use SRPT Charging to compute a lower bound on $D(OptC(I_i))$. When working with $OptC(I_i)$, we use $D_S(OptC(I_i))$ to denote the idle time incurred by canonical partial schedule $OptC(I_i)$ as computed by SRPT charging.

The basic idea is to take the profile at the start of interval I_i, either $RSRPT[I, i]$ or $OptC[I, i]$, perform SRPT on this profile assuming no jobs are released after time r_i, and then charge jobs for the idle time they incurred in the corresponding partial schedule. We first note that $RSRPT(I_{r(I)})$ and $OptC(I_{r(I)})$ can both be SRPT applied to the jobs in $RSRPT[I, r(I)]$ and $OptC[I, r(I)]$ as SRPT is optimal when there is only a single release time. Thus, we only need to worry about I_i for $1 \leq i < r(I)$.

More formally, we first order the jobs in either profile $RSRPT[I, i]$ or $OptC[I, i]$ in non-decreasing order of remaining length breaking ties arbitrarily for jobs completed in the partial schedule $RSRPT(I_i)$ or $OptC(I_i)$. 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.

We use the ordering to charge idle time as follows. Suppose job numbered j is processed for e_j time units in $RSRPT(I_i)$ or $OptC(I_i)$. We then say that jobs $j + m$, $j + 2m, \ldots$ incur e_j units of idle time and we will charge these incurred idle times to job j. We charge this way because, as we noted above, if we ran SRPT on this instance and no other jobs arrived, job j would delay exactly jobs $j + m$, $j + 2m$, etc. for e_j time units.

Example 7. Let I be the input instance from Example 6, and consider interval I_2. The profile $RSRPT[I, 2] = \langle 0, 0, 4, 4, 4, 4, 32 \rangle$. We then charge idle times as follows. Jobs 5 and 7 are delayed by job 3 for 4 time units each. Job 6 is delayed by job 4 for 4 time units. Finally, job 7 is delayed by job 5 for 4 time units. Thus, we compute $D(RSRPT(I_2)) = 16$.

The profile $OptC[I, 2] = \langle 0, 4, 4, 4, 4, 4, 28\rangle$. In $OptC(I_2)$, jobs 2 and 3 incur no idle time, jobs 4 and 5 incur 4 units of idle time each, and jobs 6 and 7 incur 6 units of idle time each. Thus, $D(OptC(I_2)) = 20$. Applying SRPT charging, we observe that $D_S(OptC(I_2)) = 20$ as well. That is, jobs 4 and 6 are delayed by job 2 for 4 units each, jobs 5 and 7 are delayed by job 3 for 4 units each, and job 6 is delayed by job 4 for 2 units while job 7 is delayed by job 5 for 2 units.

For some canonical partial schedules $OptC(I_i)$, $D_S(OptC(I_i)) < D(OptC(I_i))$.

Example 8. Let $I = \{(0, 1), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (2, 7)\}$. There are only two possible two machine canonical partial schedules for I_1. The first, denoted by $S_1(I_1)$, is the partial schedule that runs the two jobs of size 1 on one machine and one job of size 2 on the second machine. The second, denoted by $S_2(I_1)$, is to run both jobs of size 1 on the two machines in the time interval $[0, 1]$ and then the jobs of size 2 and 3 on the two machines in the time interval $[1, 2]$. $D(S_1(I_1))$ is 7; 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. However, $D_S(S_1(I_1)) = 6$. Jobs 3 and 5 are delayed by job 1 for 1 time unit each, jobs 4 and 6 are delayed by job 2 for 1 time unit each, and job 5 is delayed by job 3 for 2 time units. Thus, $D_S(S_1(I_1)) < D(S_1(I_1))$. We now show that for all canonical partial schedules $OptC(I_i)$, $D_S(OptC(I_i)) \leq D(OptC(I_i))$ so we can use $D_S(OptC(I_i))$ as a lower bound for $D(OptC(I_i))$ which is a lower bound for $D(Opt(I_i))$.

LEMMA 5.1. *For any input instance I, and any $1 \leq i \leq r(I)-1$, $D_S(OptC(I_i)) \leq D(OptC(I_i))$.*

PROOF. Let J denote the set of non-zero remaining job lengths at the beginning of the interval of I_i. Let $C(I_i)$ be the canonical partial schedule under consideration. Let J' be the set of non-zero remaining job lengths at r_{i+1}, the end of canonical partial schedule $C(I_i)$, ignoring all jobs that arrive after time r_i. We treat J and J' as input instances where all jobs are released at time 0, and the lengths of the jobs are the remaining job lengths in J and J', respectively.

The first observation is that one way to schedule input instance J is to use schedule $C(I_i)$ in the interval $[0, r_{i+1} - r_i)$ and then $SRPT(J')$ after time $r_{i+1} - r_i$. Another way to schedule input instance J is to use SRPT. Because SRPT minimizes total flow time and total idle time in cases where there is a single release time, $D(SRPT(J)) \leq D(C(I_i)) + D(SRPT(J'))$.

We can decompose $D(SRPT(J))$ into two components: (a) $D_S(C(I_i))$ which is SRPT charging of canonical partial schedule $C(I_i)$ and (b) $D(SRPT(J)) - D_S(C(I_i))$. The crucial observation is that $D(SRPT(J)) - D_S(C(I_i)) = D(SRPT(J'))$. This implies $D_S(C(I_i)) + D(SRPT(J')) = D(SRPT(J)) \leq D(C(I_i)) + D(SRPT(J'))$ which gives us the desired result that $D_S(C(I_i)) \leq D(C(I_i))$.

The reason $D(SRPT(J)) - D_S(C(I_i)) = D(SRPT(J'))$ is, because of Corollary 4.17, the relative order of remaining lengths of jobs in J' is identical to that in J. Thus, each job in J', when run in SRPT order, will delay exactly the same jobs as it did in J, and the result follows. □

Example 9. Let I be the input instance from Example 6, and consider interval I_2. The profile $OptC[I, 2] = \langle 0, 4, 4, 4, 4, 4, 28\rangle$, so the input instance $J = \{(0, 4),$

(0, 4), (0, 4), (0, 4), (0, 4), (0, 28)}. Consider the canonical partial schedule $OptC(I_2)$ that, when translated to operate on instance J, behaves as follows. On machine 1, it schedules job 1 in the interval [0, 4), job 2 in the interval [4, 5), and job 4 in the interval [5, 6). On machine 2, it schedules job 2 in the interval [0, 3) and job 3 in the interval [3, 6). It leaves an input instance $J' = \{(0, 1), (0, 3), (0, 4), (0, 28)\}$.

Now consider $SRPT(J)$. In this schedule, job 1 will delay jobs 3 and 5. Job 2 will delay jobs 4 and 6. Job 3 will delay job 5, and job 4 will delay job 6. When we consider J', job i of J' is job $i+2$ of J for $1 \leq i \leq 4$, so each job of J', when scheduled by SRPT, will delay the same jobs as it would in J when scheduled by SRPT.

More specifically, $D_S(OptC(I_2)) = 20$ because job 1 delays jobs 3 and 5 for 4 time units, job 2 delays jobs 4 and 6 for 4 time units, job 3 delays job 5 for 3 time units, and job 4 delays job 6 for 1 time unit. In comparison, $D(SRPT(J)) = 24$. Finally, $D(SRPT(J')) = 4$ as job 1 of J' delays job 3 for 1 time unit and job 2 of J' delays job 4 for 3 time units. Thus we see that $D(SRPT(J)) = D_S(OptC(I_2)) + D(SRPT(J'))$ for this example.

5.3. 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; that is $SRPT(I^s, i) \subseteq RSRPT(I, i)$ for $1 \leq i \leq r(I)$. To prove this containment property, we need the following technical results.

LEMMA 5.2. *Consider any set of jobs J all released at time 0, any number of machines m, any subset of jobs $K \subseteq J$ where $|K| \leq m$, and any time t such that SRPT completes all the jobs in J by time t. Let the jobs in K have processing times p_1 through $p_{|K|}$ where $p_i \leq p_j$ if $i < j$. Then schedule $SRPT(J - K)$ has the following property. All m machines are idle by time t, and there are distinct machines $m_1, \ldots, m_{|K|}$ such that m_i completes by time $t - p_i$.*

PROOF. SRPT is a greedy scheduling algorithm. In the case when all jobs are released at time 0, we can simulate its execution in the following way. It schedules the jobs in non-decreasing order of size placing the smallest unscheduled job (but no smaller than any job already scheduled) on the machine that currently has the smallest completion time. After this job is scheduled but before the next job is scheduled, this machine will now have the largest completion time.

Let us label the machines in $SRPT(J)$ in non-increasing order of completion time; that is, machine 1 completes last and has the most total processing time, and machine m completes first and has the least total processing time. Because of the way that SRPT schedules jobs, it follows that machine 1 receives a total of $q = \lceil J/m \rceil$ jobs including the largest job in J. We label the m largest jobs to be level q jobs, the next largest m jobs to be level $q-1$ jobs, and so on. Some machines do not have a level 1 job. Also, for $1 \leq i \leq q$, the level i job assigned to machine 1 is larger than the level i job assigned to any other machine.

We now examine what occurs when the jobs in K are removed from the input instance. It is trivial to observe that no machine will have a completion time larger than machine 1's original completion time. We now show that there will be machines with the proper holes. Assume the jobs in K are numbered from 1 to $|K|$ with $|K|$ being the largest job. We will show that for $1 \leq i \leq K$, machine i completes by time $t - p_i$.

Let $l(i)$ be the level of job i. It is trivial to observe that machine i processes at most one job from each level. The key observation is that machine i will not process any job of level $l(i)$. It will not process job i or any larger job from level $l(i)$ because i smaller jobs have been removed. That is, in place of its original level $l(i)$ job, it will select a level $l(i)+1$ job. It cannot process a smaller level $l(i)$ job because we have removed at most $i-1$ jobs within levels 1 through $l(i)-1$. That is, in place of its original job of level $j < l(i)$, it must process a different level j job. Thus, machine i's modified completion time is at most machine 1's original completion time minus the largest level $l(i)$ job. This is at most $t-p_i$ and the result follows. □

Example 10. Let $J = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ (release times ignored because all jobs are released at time 0), and suppose there are 4 machines. Machine 1 will have jobs 2, 6, and 10 for a completion time of 18. Machine 2 will have jobs 1, 5, and 9 for a completion time of 15. Machine 3 will have jobs 4 and 8 for a completion time of 12. Finally, machine 4 will have jobs 3 and 7 for a completion time of 10. Jobs 7 through 10 are level 3 jobs, jobs 3 through 6 are level 2 jobs, and jobs 1 and 2 are in level 1 jobs.

Suppose $K = \{4\}$; the job in K is a level 2 job in J. We need to show that in the resulting schedule *SRPT*$(J-K)$, the old machine 1 now completes by time $18-6=12$, where 6 is the largest level 2 job in J, and we need to show that all other machines complete by time 18. In *SRPT*$(J-K)$, the old machine 1 is assigned level 1 job 2 (as it was in *SRPT*(J)) and level 3 job 7 (it received level 2 job 6 in *SRPT*(J)). Note that this machine is not assigned any level 2 job in *SRPT*$(J-K)$, but it was assigned level 3 job 10 in *SRPT*(J). This machine has a completion time of $2+7=9 \leq 12$. In the proof, in place of receiving job 7, we consider the possibility machine one might receive level 3 job 10 and thus have a completion time of 12. The other machines all stay below 18, the original completion time of machine 1, as they receive at most 1 job from each of the 3 levels. In particular, the old machine 2 receives jobs 1, 6, and 10 for a completion time of 17 while the other machines each receive only 2 jobs.

LEMMA 5.3. *Consider any input instance I, any $1 \leq i < r(I)$, and any canonical partial schedule OptC(I_i). Let k, l, and e_j for $1 \leq j \leq l$ be the constants from the definition of RSRPT(I_i). 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^s_i.*

PROOF. Consider input instance I. Suppose at release time r_i, we release l additional jobs of size e_1 through e_l. 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 4.10, we know that *SRPT*$(I^{s'}, i+1) \leq$ *OptC*$(I', i+1)$. This implies *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 $e_l, e_{l-1}, \ldots, e_{l-z+1}$. It then follows that the $k+l-z$ shortest jobs in *SRPT*$[I^s, i]$ must be complete in *SRPT*$(I^{s'}, i+1)$.

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, by the previous lemma, there must be holes of size at least e_{l-z+1} through e_l into which jobs $\mathit{SRPT}[I^s, i][j]$ 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 5.4. *$\mathit{SRPT}(I^s, i) \subseteq \mathit{RSRPT}(I, i)$ 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 $\mathit{RSRPT}(I, 1)$ is identical to that of $\mathit{SRPT}(I^s, 1)$ 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 4.7 that $\mathit{SRPT}(I^s, i) \subseteq \mathit{RSRPT}(I, i)$ implies that $\mathit{SRPT}[I^s, i] \subseteq \mathit{RSRPT}[I, i]$.

Let k be the constant from the definition of $\mathit{RSRPT}(I_i)$. We first observe that the first k jobs in $\mathit{SRPT}(I^s, i + 1)$ must be complete or else $\mathit{SRPT}(I^s, i + 1) \not\subseteq \mathit{OptC}(I, i + 1)$. The containment property must hold after we try to assign e_j time units to jobs $\mathit{RSRPT}[I, i][k + j]$ for $1 \leq j \leq k$ as Lemma 5.3 shows that $\mathit{SRPT}(I_i^s)$ either finishes job $\mathit{SRPT}[I^s, i][k + j]$ or assigns it at least e_j units of processing. Finally, the containment property is maintained in the final stages of the RSRPT algorithm when RSRPT assigns the final X units of processing time to jobs. This is true because job $\mathit{RSRPT}[I, i][y]$ cannot be given enough processing time to make $\mathit{RSRPT}(I, i + 1)[y] < \mathit{SRPT}(I^s, i + 1)[y]$. □

We now show that RSRPT incurs at least as much idle time as stretched SRPT.

Definition 5.5. Let I be any input instance, A be SRPT or RSRPT, and $1 \leq i \leq r(I)$. We define $\mathit{Incur}(A, I, i)$ to be the idle time cost incurred by algorithm A on input instance I in the interval $[0, r_i)$. We define $\mathit{Complete}(A, I, i)$ and $\mathit{Complete}[A, I, i]$ to be the idle time cost incurred by A to finish all the jobs in profiles $A(I, i)$ and $A[I, i]$, respectively. Finally, we define $\mathit{Cost}(A, I, i) = \mathit{Incur}(A, I, i) + \mathit{Complete}(A, I, i)$ and $\mathit{Cost}[A, I, i] = \mathit{Incur}(A, I, i) + \mathit{Complete}[A, I, i]$.

Note that the total idle time cost incurred by either algorithm A is exactly $\mathit{Cost}[A, I, r(I)]$.

LEMMA 5.6. *For any input instance I and any $1 \leq i \leq r(I) - 1$, $\mathit{Cost}[\mathit{SRPT}, I, i] = \mathit{Cost}(\mathit{SRPT}, I, i + 1)$ and $\mathit{Cost}[\mathit{RSRPT}, I, i] = \mathit{Cost}(\mathit{RSRPT}, I, i + 1)$.*

PROOF. The key observation is that for A as SRPT or RSRPT, $\mathit{Incur}(A, I, i + 1) - \mathit{Incur}(A, I, i) = \mathit{Complete}[A, I, i] - \mathit{Complete}(A, I, i + 1)$. □

LEMMA 5.7. *For any profiles P_1 and P_2 where $P_1 \subseteq P_2$, $\mathit{SRPT}(P_1 \cup \{j\}) - \mathit{SRPT}(P_1) \leq \mathit{SRPT}(P_2 \cup \{j\}) - \mathit{SRPT}(P_2)$ where j is any job.*

PROOF. For any profile P, we can compute $\mathit{SRPT}(P)$ by ordering the jobs from largest to smallest where job 1 is the largest job and job $|P|$ is the smallest job. Given this ordering, jobs 1 through m each delay 0 jobs, jobs $m + 1$ through $2m$ each delay 1 job, and so on. In general, job i delays $\lfloor (i - 1)/m \rfloor$ jobs. Let j_1 be j's index when added to P_1, and let j_2 be j's index when added to P_2. In case of ties, we make j_1 and j_2 as small as possible. Because $P_1 \subseteq P_2$, $j_1 \leq j_2$. The jobs that will contribute to $\mathit{SRPT}(P_1 \cup \{j\}) - \mathit{SRPT}(P_1)$ are job j, and jobs with indices $k \geq j_1$

that are perfect multiples of m. Job j will delay $\lfloor (j_1 - 1)/m \rfloor$ jobs in $P_1 \cup \{j\}$ while it delayed no jobs in P_1. The other jobs k will each delay one extra job in $P_1 \cup \{j\}$. Likewise, the jobs that will contribute to $\mathit{SRPT}(P_2 \cup \{j\}) - \mathit{SRPT}(P_2)$ are job j, and jobs with indices $k \geq j_2$ that are perfect multiples of m. Job j will delay $\lfloor (j_2 - 1)/m \rfloor$ jobs in $P_2 \cup \{j\}$ while it delayed no jobs in P_2. The other jobs k will each delay one extra job in $P_2 \cup \{j\}$. For the jobs $k \geq j_2$ that are perfect multiples of m, since $P_1 \subseteq P_2$, the job k in P_2 increases $\mathit{SRPT}(P_2 \cup \{j\}) - \mathit{SRPT}(P_2)$ at least as much as the corresponding job k in P_1 increases $\mathit{SRPT}(P_1 \cup \{j\}) - \mathit{SRPT}(P_1)$. On the other hand, the jobs in P_1 with indices $j_1 \leq k < j_2$ that are perfect multiples of m must be no larger than j by definition of j_1. Furthermore, the number of such k plus $\lfloor (j_1 - 1)/m \rfloor$ is no larger than $\lfloor (j_2 - 1)/m \rfloor$. Thus, the contribution of these jobs plus job j to $\mathit{SRPT}(P_1 \cup \{j\}) - \mathit{SRPT}(P_1)$ is no larger than the contribution of job j to $\mathit{SRPT}(P_2 \cup \{j\}) - \mathit{SRPT}(P_2)$ and the result follows. $\square$

THEOREM 5.8. *For any input instance I, $D(\mathit{SRPT}(I^s)) \leq D(\mathit{RSRPT}(I))$.*

PROOF. The theorem can be restated as $\mathit{Cost}[\mathit{SRPT}, I^s, r(I)] \leq \mathit{Cost}[\mathit{RSRPT}, I, r(I)]$. It is easy to see that $\mathit{Cost}(\mathit{SRPT}, I^s, 1) = \mathit{Cost}(\mathit{RSRPT}, I, 1) = 0$. For $1 \leq i \leq r(I)$, Corollary 5.4, Fact 4.6 and Lemma 5.7 imply that $\mathit{Cost}[\mathit{RSRPT}, I, i] - \mathit{Cost}(\mathit{RSRPT}, I, i) \leq \mathit{Cost}[\mathit{SRPT}, I^s, i] - \mathit{Cost}(\mathit{SRPT}, I^s, i)$. That is, introducing the jobs released at time r_i in I increases the Complete cost of RSRPT more than introducing the jobs released at time r_i^s in I^s increases the Complete cost of SRPT. This means that if we have $\mathit{Cost}(\mathit{SRPT}, I^s, i) \leq \mathit{Cost}(\mathit{RSRPT}, I, i)$, then we get $\mathit{Cost}[\mathit{SRPT}, I^s, i] \leq \mathit{Cost}[\mathit{RSRPT}, I, i]$. Combining this observation with Lemma 5.6 allows us to go from $\mathit{Cost}(\mathit{SRPT}, I^s, 1) = \mathit{Cost}(\mathit{RSRPT}, I, 1)$ to the desired $\mathit{Cost}[\mathit{SRPT}, I^s, r(I)] \leq \mathit{Cost}[\mathit{RSRPT}, I, r(I)]$. $\square$

5.4. OPT INCURS AT LEAST AS MUCH IDLE TIME AS RSRPT.

In this section, let k, q, l and e_j for $1 \leq j \leq l$ be the constants from the definition of $\mathit{RSRPT}(I_i)$.

LEMMA 5.9. *For any $1 \leq i \leq r(I)$, $\mathit{RSRPT}(I, i) \leq \mathit{OptC}(I, i)$.*

PROOF. We prove this by induction on i. The base case with $i = 1$ is trivially true as the profiles are both empty. Assume the result holds for $1 \leq i < r(I)$. We now need to show the result holds for $i + 1$. Transitivity along with $\mathit{RSRPT}(I, i) \leq \mathit{OptC}(I, i)$ and $\mathit{OptC}(I, i) \leq \mathit{Opt}(I, i)$ imply that $\mathit{RSRPT}(I, i) \leq \mathit{Opt}(I, i)$. Corollary 4.7 yields $\mathit{RSRPT}[I, i] \leq \mathit{Opt}[I, i]$.

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 $\mathit{Opt}[I, i]$ must be Y smaller than the prefix sum of the first k jobs in $\mathit{RSRPT}[I, i]$, but this is not possible because $\mathit{RSRPT}[I, i] \leq \mathit{Opt}[I, i]$. This also proves that the value X in the construction of $\mathit{RSRPT}(I_i)$ must be non-negative.

Suppose that $\mathit{RSRPT}(I, i+1)$ is not less than $\mathit{OptC}(I, i+1)$. Then there must be a smallest integer j such that $\mathit{RSRPT}^j(I, i+1) > \mathit{OptC}^j(I, i+1)$. We now show that no such j exists. We first examine how RSRPT doles out the X extra processing time it gains within the first $k + q$ jobs. Specifically, the rules will be applied in the order given. That is, some consecutive set of jobs will be completed, then some consecutive set of jobs will be run until they have received as much total processing as the corresponding job will have received in stretched SRPT's schedule, and finally at most one job will receive the remaining surplus amount of processing time. This means in $RSRPT(I, i+1)$, the first $a \geq k+q$ jobs are all complete, the next b jobs all have the same remaining length as the corresponding jobs in $SRPT(I^s, i+1)$, and one final job receives some extra processing time. Clearly, $j > a$ as $RSRPT^a(I, i+1) = 0$. Likewise, $j > a+b$ as $RSRPT^{a+b}(I, i+1) = SRPT^{a+b}(I^s, i+1)$, and we know that $SRPT(I^s, i+1) \leq OptC(I, i+1)$. Finally, we will argue that j cannot be greater than $a+b$ which shows that j cannot exist.

Consider any $u \geq a+b+1$. By our inductive hypothesis, $RSRPT^u[I, i] \leq OptC^u[I, i]$. Furthermore, by the definition of u, a, b, and RSRPT, RSRPT must devote at least as much processing time to the first u jobs of $RSRPT(I, i+1)$ as OptC does. These two facts combine to show that $RSRPT^u(I, i+1) \leq OptC^u(I, i+1)$ and the result follows. □

This leads to the following result.

COROLLARY 5.10. *For any input instance I and any $1 \leq i \leq r(I)$, the idle time incurred by partial schedule $D(RSRPT(I_i)) \leq D_S(OptC(I_i))$.*

PROOF. The key observation is that for any integer x, RSRPT devotes less total processing time in interval I_i to the first x jobs in $RSRPT[I, i]$ than $OptC$ does to the first x jobs in $Opt[I, i]$. This clearly holds for $x \leq k$ as both RSRPT and $OptC$ complete the smallest k jobs and $RSRPT[I, i] \leq Opt[I, i]$. This also holds for $x \leq k+q$ because for $k+1 \leq j \leq k+q$, RSRPT uses at most e_j processing time to complete the jth smallest job in I_i while $OptC$ devotes e_j time to the jth smallest job in I_i. This also holds for $x > k+l$ as we ensure that $RSRPT(I_i)$ does no more processing than $OptC(I_i)$ does and $OptC(I_i)$ devotes all its processing to the first $k+l$ jobs in $OptC[I, i]$.

We now show this holds for $k+q < x \leq k+l$. For $k+q < j \leq k+l$, we ensure that RSRPT processes job j in $RSRPT[I, i]$ for at least e_j time units. In particular, for $x < j \leq k+l$ RSRPT processes each job j for at least as much time as $OptC$ does. Thus, the total time spent by RSRPT on the jobs up to x cannot exceed the total time spent by $OptC$ on the jobs up to x.

Given that SRPT charging more heavily charges the smaller jobs that run, this observation implies the desired result. □

COROLLARY 5.11. *For any input instance I, $D(RSRPT(I)) \leq D(Opt(I))$.*

PROOF. This result immediately follows from the previous corollary. □

THEOREM 5.12. *For any input instance I, m available machines, and $s \geq 2 - 1/m$, $D(SRPT(I^s)) \leq D(Opt(I))$ and $F(SRPT(I^s)) \leq F(Opt(I))$.*

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 nonmigratory algorithms also have this property? We have shown that existing nonmigratory algorithms are not s-speed $1/s$-competitive algorithms for any $s \geq 1$. 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.

ACKNOWLEDGMENT. The authors thank the anonymous referee for his or her very helpful comments. They significantly improved the clarity of our presentation.

REFERENCES

ANDERSON, E., AND POTTS, C. 2004. On-line scheduling of a single machine to minimize total weighted completion time. *Math. Oper. Res. 29*, 686–697.

AVRAHAMI, N., AND AZAR, Y. 2003. Minimizing total flow time and total completion time with immediate dispatching. In *Proceedings of the 15th ACM Symposium on Parallel Algorithms and Architectures*. ACM, New York, 11–18.

AWERBUCH, B., AZAR, Y., LEONARDI, S., AND REGEV, O. 2001. Minimizing the flow time without migration. *SIAM J. Comput. 31*, 1370–1382.

BUSSEMA, C., AND TORNG, E. 2006. Greedy multiprocessor server scheduling. *Oper. Res. Letters 34*, 451–458.

CHEKURI, C., GOEL, A., KHANNA, S., AND KUMAR, A. 2004. Multi-processor scheduling to minimize flow time with ϵ-resource augmentation. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*. ACM, New York, 363–372.

CHEKURI, C., KHANNA, S., AND ZHU, A. 2001. Algorithms for weighted flow time. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*. ACM, New York, 84–93.

CONWAY, R. W., MAXWELL, W. L., AND MILLER, L. W. 1967. *Theory of Scheduling*. Addison-Wesley, Reading, MA.

COULSTON, C., AND BERMAN, P. 1999. Speed is more powerful than clairvoyance. *Nord. J. Comput. 6*, 181–193.

EDMONDS, J. 2000. Scheduling in the dark. *Theoret. Comput. Sci. 235*, 109–141.

GRAHAM, R. L., LAWLER, E. L., LENSTRA, J. K., AND RINNOOY KAN, A. H. G. 1979. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Ann. Disc. Math. 5*. 287–326.

KALYANASUNDARAM, B., AND PRUHS, K. 2000. Speed is as powerful as clairvoyance. *J. ACM (JACM) 47*, 617–643.

LEONARDI, S. 2003. A simpler proof of preemptive flow-time approximation. In *Approximation and On-line Algorithms*. Lecture Notes in Computer Science. Springer-Verlag, New York.

LEONARDI, S., AND RAZ, D. 1997. Approximating total flow time on parallel machines. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*. ACM, New York, 110–119.

PHILLIPS, C., STEIN, C., TORNG, E., AND WEIN, J. 2002. Optimal time-critical scheduling via resource augmentation. *Algorithmica 32*, 163–200.

SCHRAGE, L. E. 1968. A proof of the optimality of the shortest remaining processing time discipline. *Oper. Res. 16*, 678–690.

RECEIVED APRIL 2006; REVISED NOVEMBER 2006 AND JUNE 2007; ACCEPTED NOVEMBER 2007