SAND97-0809C
SAND--97-0809C

# Optimal Time-Critical Scheduling Via Resource Augmentation

(Extended Abstract)

CONF-970599--1-Exc. Abst.

Cynthia A. Phillips * Cliff Stein † Eric Torng ‡ Joel Wein§


## Abstract

We consider two fundamental problems in dynamic scheduling: scheduling to meet deadlines in a preemptive multiprocessor setting, and scheduling to provide good response time in a number of scheduling environments. When viewed from the perspective of traditional worst-case analysis, no good on-line algorithms exist for these problems, and for some variants no good off-line algorithms exist unless $\mathcal{P} = \mathcal{NP}$.

We study these problems using a relaxed notion of competitive analysis, introduced by Kalyanasundaram and Pruhs, in which the on-line algorithm is allowed more resources than the optimal off-line algorithm to which it is compared. Using this approach, we establish that several well-known on-line algorithms, that have poor performance from an absolute worst-case perspective, are optimal for the problems in question when allowed moderately more resources. For the optimization of average flow time, these are the first results of any sort, for any $\mathcal{NP}$-hard version of the problem, that indicate that it might be possible to design good approximation algorithms.



* caphill@cs.sandia.gov. Sandia National Labs, Albuquerque, NM. This work was supported in part by the United States Department of Energy under Contract DE-AC04-94AL85000.

† cliff@cs.dartmouth.edu. Department of Computer Science, Sudikoff Laboratory, Dartmouth College, Hanover, NH. Research partially supported by NSF Award CCR-9308701 and NSF Career Award CCR-9624828. Some of this work was done while this author was visiting Stanford University, and while visiting the first author at Sandia National Laboratories.

‡ torng@cps.msu.edu. Department of Computer Science, A-714 Wells Hall, Michigan State University, East Lansing, MI 48824-1027. Some preliminary work done while the author was a Stanford graduate student, supported by a DOD NDSEG Fellowship, NSF Grant CCR-9010517, Mitsubishi Corporation, and NSF YI Award CCR-9357849, with matching funds from IBM, Schlumberger Foundation, Shell Foundation and Xerox Corporation.

§ wein@mem.poly.edu. Department of Computer Science, Polytechnic University, Brooklyn, NY, 11201. Research partially supported by NSF Research Initiation Award CCR-9211494, NSF Grant CCR-9626831, and a grant from the New York State Science and Technology Foundation, through its Center for Advanced Technology in Telecommunications.


MASTER

## 1 Introduction

In this paper, we consider two fundamental multiprocessor scheduling problems:

- on-line multiprocessor scheduling of sequential jobs in a hard-real-time environment, in which all jobs must be completed by their deadlines, and
- on-line multiprocessor scheduling of sequential jobs to minimize average flow time (average response time) in preemptive and nonpreemptive settings.

These problems have defied all previous (worst-case) analytic attempts to identify effective on-line algorithms for solving them. For example, Dertouzos and Mok proved that no on-line algorithm can legally schedule all feasible input instances of the hard-real-time scheduling problem for $m \geq 2$ machines [3]. Furthermore, there is no obvious notion of an approximation algorithm[1] for this problem since all jobs must be completed. For the various versions of the flow time problem, while approximations are acceptable, a variety of results [13, 16] show that no on-line algorithm can guarantee a reasonable approximation ratio. The net result is that the traditional worst-case analysis techniques have failed in two ways:

- they have failed to identify effective algorithms for these problems, and
- they have failed to differentiate between algorithms whose performance is observed empirically to be rather different.

In this paper we give the first encouraging results that apply worst-case analysis to these two multiprocessor problems. We utilize a new method of analysis, introduced by Kalyanasundaram and Pruhs [11] (for one-processor scheduling) of comparing the performance of an on-line algorithm

[1] There has been significant work in the area of best-effort real-time scheduling, in which one tries to maximize the total weight of the jobs scheduled by their deadlines, but this is not really an appropriate approximation for hard-real-time scheduling since the fundamental assumption of best-effort scheduling is that it is acceptable for jobs to not complete by their deadlines. Furthermore, even if one accepts best-effort real-time scheduling as a reasonable way to approximate hard-real-time scheduling, Koren et al. showed that the best competitive ratio any on-line algorithm can achieve is lower bounded by $(\frac{\Delta}{\Delta-1})m(\Delta^{1/m}-1)$ where $m$ is the number of machines and $\Delta$ is the ratio between the weights of the most important and least important jobs in the input instance [14].

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

# DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, make any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

**DISCLAIMER**

**Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.**

to the performance of an optimal off-line algorithm when the on-line algorithm is given extra resources. For example, in a preemptive multiprocessor environment, we show that when given machines that are twice as fast, the *shortest-remaining-processing-time* algorithm gives *optimal* performance for average flow time, and the *earliest-deadline-first* and *least-laxity-first* algorithms, give optimal performance for meeting deadlines. In the nonpreemptive setting, we also show that simple greedy algorithms perform well for optimizing average flow time when given more machines. Many of our results for average flow time extend to average *weighted flow time* as well. While we analyze our algorithms using the metric introduced in [11], our results differ from that work in two fundamental ways:

- we consider the multiprocessor case, and
- we almost always find schedules whose objective function value is *optimal*, rather than near-optimal.

In addition, our results are in a somewhat different on-line model. We assume that when a job arrives, its processing time is known; we discuss our model in more detail later in this introduction.

We feel that our results have two practical implications. First, our results provide system designers with analytic guidelines for coping with lack of knowledge of the future. For example, our results that describe the performance of an algorithm when given extra machines tell a system designer how many extra processors are needed to insure a desired performance level. Our results that describe the performance of an algorithm when given faster machines not only tell a system designer how much faster the processors need to be to insure a desired performance level, they also have implications for the performance of the original system in a setting where job-arrival rate is reduced. In particular, we show that, for the problem of minimizing the average flow time, on-line algorithms can achieve small constant competitive ratios with respect to offline algorithms given *identical* resources if the arrival rate of jobs in the on-line system is somewhat slower than the arrival rate of jobs in the identical offline system. Thus reduced job-arrival rate is an "extra resource" that can compensate for lack of knowledge of the future, much as the hardware-based increased speed or increased machines can. Second, more speculatively, if an algorithm, when allowed a bit more speed or resources, performs well on all input instances, then perhaps the instances on which it performs very poorly under traditional worst-case analysis have a special structure, and it is possible that such instances may be less likely to arise in practice. We suggest, though, that the ultimate decision of whether this sort of analysis is meaningful will depend on the sorts of algorithms the analysis recommends and whether or not it yields interesting and new distinctions between algorithms. When evaluated from this perspective, our results provide powerful evidence that "extra-resource" analysis is a useful tool in the analysis of on-line scheduling problems.

Although our notion of worst-case analysis is not the traditional one, these analytic results for on-line algorithms are the first that provide any evidence that it is possible to design polynomial-time off-line algorithms for the minimization of average flow time with performance guarantees of good quality for any $\mathcal{NP}$-hard variant of the problem.

**Problem Definitions:** We are given $m$ identical parallel machines and an input instance (job set) $I$, which is a collection of $n$ independent jobs $\{J_1, J_2, \ldots, J_n\}$. Each job $J_j$ has a release date $r_j$, an execution time (also referred to as length or processing time) $p_j$, possibly a weight $w_j$, and, in the real-time setting only, a deadline $d_j$. The ratio of the length of the longest job to that of the shortest job in instance $I$ is denoted $\Delta(I)$, or simply $\Delta$ when the job set $I$ is unambiguous. For any input instance $I$, we let $I^l$ denote the *l-stretched* input instance where job $J_i$ has release time $lr_j$ instead of $r_j$. A job can run on only one machine at a time and a machine can process only one job at a time. We will often denote the completion time of job $J_j$ in schedule $S$ by $C_j^S$ and will drop the superscript when the schedule is understood from context. The *flow time*, or *response time* of a job in schedule $S$ is $F_j^S \equiv C_j^S - r_j$; the *total flow time* of schedule $S$ is $\sum_j F_j^S$, whose minimization is equivalent to the minimization of *average flow time* $\frac{1}{n}\sum F_j$. If the jobs have weights, we can also define the *total weighted flow time* of schedule $S$ by $\sum_j w_j F_j^S$, or equivalently the *average weighted flow time* $\frac{1}{n}\sum_j w_j F_j$. For hard real-time scheduling with deadlines, we say a schedule $S$ is *optimal* if each job $J_j$ is scheduled by its deadline $d_j$, i.e. $C_j^S \leq d_j$. These are often called *feasible* schedules.

We will consider both preemptive and nonpreemptive scheduling models. In a preemptive scheduling model, a job may be interrupted and subsequently resumed with no penalty; in a nonpreemptive scheduling model, a job must be processed in an uninterrupted fashion.

**Our On-line Model and Methods of Analysis:** We consider *on-line* scheduling algorithms which construct a schedule in time, and must construct the schedule up to time $t$ without any prior knowledge of jobs that will become available at time $t$ or later. When a job arrives, however, we assume that all other relevant information about the job is known; this model has been considered by many authors, e.g. [8, 9, 10, 22, 23], and is a reasonable model of a number of settings from repair shops to timesharing on a supercomputer. (For example, in the latter setting, when one submits a job to a national supercomputer center, one must give an estimate of the job size.)

We will analyze our algorithms by considering their performance when they are allowed to run on more and/or faster machines as well as when they are run on $l$-stretched input instances. Given an input $I$ to a scheduling problem with $m$ machines and (optimal) objective function value $V$, an *s-speed ρ-approximation algorithm* finds a solution of value $\rho V$ using $m$ speed-$s$ machines. A *w-machine ρ-approximation algorithm* finds a solution of value $\rho V$ using $wm$ machines. An *l-stretch* $\rho$-approximation finds a solution to $I^l$ of value $\rho V$ using machines identical in number and speed to the offline algorithm. For a problem with deadlines, we consider $V$ to be the objective of scheduling all jobs by their deadlines, so it is always the case that $\rho = 1$. In fact, for both deadlines and flow-time, we will usually be concerned with the case where $\rho = 1$, so we will omit the term "$\rho$-approximation" when this is true.

**Results – Preemptive Problems:** We now discuss our results, which are summarized in Figure 1. We first study on-line preemptive scheduling for both objectives: minimum total flow time and hard real-time scheduling with deadlines. We show that several simple and widely-used scheduling heuristics (for which worst-case analysis yields a pessimistic evaluation) are $(2 - \frac{1}{m})$-speed algorithms. These results follow from a general result that characterizes the amount of work done by any "busy" algorithm (one that never allows any unforced idle time) run at speed $s$ when compared to that done by *any* algorithm run at speed 1. We also show that no $(1+\epsilon)$-speed algorithms exist for either problem for

| | Speed to Achieve Optimal | Extra Machines to Achieve Optimal |
|---|---|---|
| hard deadline | $\frac{6}{5} < s \leq 2 - \frac{1}{m}$ | $\frac{5}{4} < w$<br>$c_{\text{all}} < w(LLF) \leq c \log \Delta$<br>$c\Delta < w(EDF)$ |
| preemptive, $\sum F_j$ | $\frac{22}{21} < s \leq 2 - \frac{1}{m}$ | |
| preemptive, $\sum w_j F_j$, $m = 1$ | $s \leq 2$ | |
| nonpreemptive, $\sum w_j F_j$ | | $w \leq c \log \Delta$<br>$w \leq c \log n$ [$1 + o(1)$-approx] |

Figure 1: Summary of main algorithms and hardness results. The notation $x < s \leq y$ means the problem can be solved with speed-$y$ machines, but cannot be solved optimally with speed-$x$ machines. Similarly $w$ is for $w$-machine algorithms, and $w$(EDF) is the number of extra machines given to earliest-deadline-first algorithm. LLF is the least-laxity first algorithm. We use $c$ to denote some constant and $c_{\text{all}}$ to denote all constants. $\Delta$ is the ratio of the longest length to the shortest length.

small $\epsilon$ (1/5 for meeting deadlines and 1/21 for flow time).

More specifically, for preemptive hard real-time scheduling, we analyze two simple and widely used on-line algorithms, earliest-deadline-first (EDF) [2] and least-laxity-first (LLF) [18]. At time $t$ in an $m$-processor system, EDF schedules the $m$ jobs currently in the system which have the earliest deadlines while LLF schedules the $m$ jobs currently in the system which have the smallest laxities (at time $t$, a job $J_j$ has *laxity* $(d_j - t) - (p_j - x_j)$ where $x_j$ is the amount of processing $J_j$ received prior to time $t$). In the uniprocessor setting ($m = 1$), both EDF [2] and LLF [18] can schedule any feasible input instance. In the multiprocessor environment ($m \geq 2$), no on-line algorithm legally schedules all feasible $m$-machine input instances [3]. Nonetheless, EDF and LLF are likely heuristic choices in practice with EDF being simpler to implement and LLF more effective in general.

In the faster-machine model, we show that both EDF and LLF are exactly $(2 - \frac{1}{m})$-speed algorithms for the problem of hard-real-time scheduling; this provides some theoretical justification for these heuristic choices. We also show that no $(1 + \epsilon)$-speed algorithm exists for this problem for $\epsilon < 1/5$ and $m \geq 2$. In the extra-machine model, we show that LLF is an $O(\log \Delta)$-machine algorithm while EDF is not a $o(\Delta)$-machine algorithm for any $m \geq 2$. We note that our analysis of LLF is fairly tight by showing that LLF is not a $c$-machine algorithm for any constant $c$. We also show that no $(1 + \epsilon)$-machine algorithm exists for this problem for $\epsilon < 1/4$ for $m \geq 2$. Comparing these results with those in the faster-machine model, we see a contrast between the power of extra machines and the power of extra speed in the preemptive setting.

For the problem of minimizing total flow time, we analyze the simple and widely used SRPT (Shortest Remaining Processing Time) Rule which always schedules the $m$ jobs with the shortest remaining processing time. In the uniprocessor setting ($m = 1$), SRPT is optimal. In the multiprocessor setting ($m \geq 2$), however, SRPT is only an $O(\log m)$-approximation [16]. Furthermore, Leonardi and Raz have shown that no randomized on-line algorithm can be any better than an $\Omega(\log m)$- approximation algorithm [16]. In contrast, we show that SRPT is a $(2 - \frac{1}{m})$-speed algorithm. Thus, we provide encouraging theoretical evidence to support the use of SRPT in practice. We also show that no $(1 + \epsilon)$-speed algorithm exists for this problem for $\epsilon < 1/21$ for $m \geq 2$.

We also consider the problem of scheduling a single machine preemptively to minimize average *weighted* flow time. This weighted variant reflects the fact that in many settings, the jobs to be scheduled have different priorities. In contrast to the unweighted problem, scheduling preemptively on a single machine to optimize average weighted flow time is $\mathcal{NP}$-hard [15]. Recently there has been much progress in developing off-line approximation algorithms for the related $\mathcal{NP}$-hard problem of minimizing $\sum_j w_j C_j$ via certain classes of linear programming relaxations [1, 9, 20], but no non-trivial polynomial-time approximation algorithms are known for $\sum w_j F_j$. (Since $\sum w_j C_j = \sum w_j F_j + \sum w_j r_j$, an optimal schedule for $\sum w_j C_j$ is also an optimal schedule for $\sum w_j F_j$, but a $\rho$-approximation for $\sum w_j C_j$ may be a very poor approximation for $\sum w_j F_j$.) We show how to use the linear-programming relaxations considered by [9] to develop an (on-line) 2-speed algorithm for this problem.

**Nonpreemptive Models:** We also consider the problem of scheduling nonpreemptively to minimize average weighted flow time in both the uniprocessor and multiprocessor settings. Note that the simplest possible variant of this problem (off-line, uniprocessor, unweighted) is already a difficult problem with very strong nonapproximability results. In particular, Kellerer et al. [13] recently showed that there exists no polynomial-time $o(\sqrt{n})$-approximation algorithm for this problem unless $\mathcal{P} = \mathcal{NP}$, and Leonardi and Raz have given analogous hardness results for parallel machines [16]. Thus, traditional worst-case analysis has little to offer to practitioners. In sharp contrast to these results, we give an $O(\log \Delta)$-machine algorithm and an $O(\log n)$-machine $(1+o(1))$-approximation algorithm for the on-line minimization of total weighted flowtime on parallel machines. These results generalize further to nonpreemptive scheduling to meet due dates.

We then offer some evidence that indicates it may be difficult to improve upon our results. A common method of analyzing the performance of a nonpreemptive algorithm is with respect to the optimal preemptive solution, which is an obvious lower bound [20, 13]. Let $S_p$ be the schedule with optimal flowtime in the preemptive $m$-machine setting for some instance $I$, and let $S_w$ be the schedule with optimal flowtime in the nonpreemptive, $wm$-machine setting. We give a polynomial-size lower bound on the gap between $\sum_j F_j^{S_p}$ and $\sum_j F_j^{S_w}$ for any constant $w \geq 2$, even if the $wm$ machines are speed-2. Thus the analysis of any $O(1)$-machine 2-speed non-preemptive flow-time algorithm will require a stronger lower bound than the flow-time of the preemptive 1-machine 1-speed schedule. In contrast to this negative result about $O(1)$-machine algorithms, we give $O(\log n)$-machine 2-speed algorithm that nonpreemptively achieves the polynomially-loose 1-machine 1-speed preemptive lower bound.

**Speed and Stretch:** We conclude by showing that any $s$-speed $\rho$-approximation algorithm is also an $s$-stretch $\rho s$-approximation algorithm when we consider the problems of minimizing average flow time or average weighted flow time. In light of this result, throughout the paper we focus only on analyzing faster machines and extra machines, but remember that the flow-time results for faster machines extend to stretched schedules.

**Previous Results:** The only previous work on this mode of analysis of which we are aware is the work of Kalyanasundaram and Pruhs who introduced it [11]. They studied the minimization of preemptive total flow time and best-effort firm-real-time scheduling. Their flow time work differs from ours in several important characteristics. They studied only the unweighted uniprocessor[2] setting where the algorithm has no knowledge of $p_j$ until job $J_j$ completes. For this problem, they were able to show a simple on-line algorithm was an $s$-speed $\left(1+\frac{1}{s-1}\right)$-approximation algorithm for minimizing flow time. (Note our relationship between faster machines and stretched schedules implies that this algorithm is also an $s$-stretch $(s+\frac{s}{s-1})$-approximation algorithm.) This result is quite dramatic as it was previously shown that no deterministic on-line algorithm can approximate the optimal flow time within a factor of $\Omega(n^{\frac{1}{3}})$ [19]. However, their analysis does not yield an $s$-speed on-line algorithm for minimizing average flow time for any $s$. At speed 2, for example, they only guarantee a 2-approximation. Furthermore, in our model of on-line scheduling in which processing requirements are known, this problem is a simple one that can be solved optimally by SRPT.

Their real-time problem differs from ours in two significant ways. First, they assume that jobs have values and the goal is to maximize the sum of the values of completed jobs; that is, it is acceptable for jobs to not complete by their deadlines. Second, they derive results only for the uniprocessor setting. Again, in contrast to quite strong traditional on-line lower bounds, they were able to show that an on-line scheduling algorithm was an $s$-speed $\left(1+\frac{2}{s-1}\right)$-approximation algorithm, for $s>1$; at speed 2, this corresponds to a 3-approximation.

## 2 On-line preemptive scheduling using faster machines

In this section, we show that SRPT is a $(2-1/m)$-speed algorithm for preemptively minimizing average flow time and that EDF and LLF are $(2-1/m)$-speed algorithms for hard-real-time scheduling. We also show that no $(1+\epsilon)$-speed algorithm exists for the hard-real-time scheduling problem for $\epsilon<1/5$, and that no $(1+\epsilon)$-speed algorithm exists for minimizing average flow time for $\epsilon<1/21$.

**Theorem 2.1** *SRPT is a $(2-\frac{1}{m})$-speed algorithm for the preemptive minimization of average flow times on parallel machines.*

**Theorem 2.2** *EDF and LLF are $(2-\frac{1}{m})$-speed algorithms for the preemptive scheduling of jobs with deadlines on parallel machines.*

The key to the proof of both theorems is the following fundamental relationship between machine speed and total work done by any busy algorithm such as SRPT or EDF. We let $A(j,t)$ denote the amount of processing algorithm $A$ specifies for job $J_j$ by time $t$. For job set $J$ let $A(J,t)=\sum_{j\in J}A(j,t)$.

**Lemma 2.3** *Consider any input instance $I$, any time $t$, any $m\geq 1$, and any $1\leq\beta\leq(2-\frac{1}{m})$. Define $\alpha=\frac{2-1/m}{\beta}$. For any busy scheduling algorithm $A$ using $m$ speed-$\alpha$ machines, $A(I,\beta t)\geq A'(I,t)$ for any algorithm $A'$ using $m$ speed-1 machines.*

*Proof Sketch:* We will prove this by contradiction. Fix an input instance $I$, and let $t$ be the smallest time such that $A(I,\beta t)\leq A'(I,t)$. Since $A$ has done less total work than $A'$, $A$ must have done less work on some specific job $J_j$. For $A$, consider the time period from time $r_j$ to $\beta t$; for $A'$, consider the time period from $\frac{r_j}{\beta}$ to $t$. From time $r_j$ to time $\beta t$, there are two types of time intervals: overloaded intervals where more than $m$ jobs are available to be scheduled by $A$ and underloaded time periods where at most $m$ jobs are available to be scheduled. Let the total length of the overloaded time intervals be $x$ and the total length of the underloaded time intervals be $y$.

By the definition of time $t$, we know that $A(I,r_j)\geq A'(I,\frac{r_j}{\beta})$. Therefore, we can conclude that the total work done by $A$ from time $r_j$ to time $\beta t$ must be strictly less than the total work done by $A'$ from time $\frac{r_j}{\beta}$ to time $t$. During overloaded intervals, $A$ uses all $m$ machines while during underloaded intervals, $A$ uses at least one machine to run job $J_j$. Therefore, $A$ does at least $\alpha mx+\alpha y$ total work from time $r_j$ to time $\beta t$. Algorithm $A'$ does at most $m(t-\frac{r_j}{\beta})$ work from time $\frac{r_j}{\beta}$ to $t$ since $A'$ only has $m$ speed-1 machines. Thus, we conclude that

$$m\left(t-\frac{r_j}{\beta}\right)>\alpha(mx+y). \tag{1}$$

By the definition of job $J_j$, we know the work done by $A$ on $J_j$ from time $r_j$ to time $\beta t$ must be strictly less than the work done by $A'$ from time $\frac{r_j}{\beta}$ to time $t$. Algorithm $A$ does at least $\alpha y$ work on $J_j$ from time $r_j$ to time $\beta t$, while algorithm $A'$ does at most $t-\frac{r_j}{\beta}$ work on $J_j$ from time $\frac{r_j}{\beta}$ to time $t$. Thus, we conclude that

$$t-\frac{r_j}{\beta}>\alpha y. \tag{2}$$

We now show that (1) and (2) cannot simultaneously be true. Add (1) to $(m-1)$ times (2) to obtain

$$\left(t-\frac{r_j}{\beta}\right)(2m-1)>\alpha m(x+y).$$

Recall that $x+y=\beta t-r_j$, and so if we divide the left side by $\beta t-r_j$ and the right by $x+y$, we obtain

$$\begin{aligned}\frac{2m-1}{\beta} &> \alpha m\\ \Leftrightarrow 2-\frac{1}{m} &> \alpha\beta.\end{aligned}$$

But this last inequality is a contradiction, since by definition $\alpha\beta=2-\frac{1}{m}$. □

*Proof of Theorem 2.1:* We will actually prove the following more general result. Consider any input instance $I$. Let $S(I)$ be any legal schedule for $I$ using $m$ speed-1 machines and $SRPT(I)$ be the schedule derived by applying SRPT

[2] In their extended abstract [11] they claim parallel machines results but have retracted this claim [12].

to $I$ using $m$ speed-$(2-1/m)$ machines. Then for any time $t$, the number of completed jobs in $SRPT(I)$ is at least as many as the number of completed jobs in $S(I)$. This implies that for all $k$, the $k$th job completed in $SRPT(I)$ finishes no later than the $k$th job completed in $OPT(I)$ (a specific legal schedule), so Theorem 2.1 clearly follows.

We prove this general result as follows. Define $I_t \subseteq I$ to be the set of jobs that complete by time $t$ in schedule $S(I)$. By Lemma 2.3 and the definition of $I_t$, all $|I_t|$ jobs will be completed by time $t$ in $SRPT(I_t)$. A lemma from [20] proves that if SRPT completes $k$ jobs of input instance $I'$ by time $t$ and $I' \subseteq I''$, then it completes at least $k$ jobs of input instance $I''$ by time $t$. We apply this result with $I_t = I'$ and $I = I''$ to conclude that for any time $t$, at least $|I_t|$ jobs will be completed by time $t$ in $SRPT(I)$. □

*Proof Theorem 2.2:* Consider any input instance $I$ which can be legally scheduled on $m$ speed-1 machines. Rename the jobs in order of deadlines so that $d_i \leq d_j$ for $1 \leq i < j \leq n$. Define $I_t$ to be the set of jobs $\{J_1, \ldots, J_t\}$. We prove by induction on $t$ that EDF, using $m$ speed-$(2-1/m)$ machines, legally schedules $I_t$.

Clearly EDF legally schedules $I_1$. Assume EDF legally schedules $I_t$ for $t \geq 1$. Consider what EDF does with $I_{t+1}$. Let $Q \subseteq I_{t+1}$ denote the set of jobs which have deadline $d_{t+1}$ (note this set may contain only job $J_{t+1}$). Because $EDF$ gives priority to jobs with earlier deadlines, we know that $EDF(I_t)$ is identical to $EDF(I_{t+1})$ when we ignore jobs in $Q$ in both schedules. Thus, all jobs in $I_{t+1} - Q$ complete by their deadlines in $EDF(I_{t+1})$. By Lemma 2.3, EDF has done at least as much work as OPT on $I_{t+1}$ by time $d_{t+1}$. In particular, since OPT has completed all jobs in $I_{t+1}$ by time $d_{t+1}$, this means EDF has completed all jobs in $Q$ by time $d_{t+1}$. Thus, EDF legally schedules $I_{t+1}$.

Thus, by the principle of induction, EDF is a speed-$(2-1/m)$ algorithm. The fact that LLF is a speed-$(2-1/m)$ algorithm follows from Leung's result that LLF can legally schedule any instance EDF can legally schedule given the same number of processors with identical capabilities [17]. □

We observe that Theorem 2.2 is tight as we can show that both EDF and LLF are not speed $2 - \frac{1}{m} - \epsilon$ algorithms for any $\epsilon > 0$. This raises the question of whether there are algorithms which are $c$-speed algorithms for some $c < 2$. We cannot completely resolve this question, but we do show that no $(1+\epsilon)$-speed algorithm exists for either problem for small $\epsilon$.

**Theorem 2.4** *There is no on-line $(1+\epsilon)$-speed algorithm for scheduling with deadlines on $m$ machines, $m \geq 2$ and $m$ even, for $\epsilon < 1/5$, and there is no on-line $(1+\epsilon)$-speed algorithm for scheduling to minimize average flow time on $m$ machines, $m \geq 2$ and $m$ even, for $\epsilon < 1/21$.*

*Proof Sketch:* For the first part of the theorem, we give a simple proof for $\epsilon = 1/9$ and sketch how to extend the result to $\epsilon = 1/5$. The proof for $\epsilon = 1/9$ is based on the following adversary strategy. At time 0, $m$ jobs of size 1 with deadlines at time 2 and $m/2$ jobs of size 2 with deadlines at time 4 are released. If an on-line algorithm run with $m$ speed-$(1+\epsilon)$ machines does at most $2m/9$ units of work on jobs of size 2 from time 0 to time 1, then $m$ jobs of size 2 with deadlines at time 4 are released at time 2 (scenario A). Otherwise, $m$ jobs of size 1 with deadlines at time 2 are released at time 1 (scenario B).

In scenario A, the algorithm does at most $2m/9$ units of work on jobs of size 2 from time 0 to time 1, and at most $m(1+\epsilon)/2$ units of work on size-2 jobs from time 1 to time 2, leaving at least $5m/18 - m\epsilon/2$ units of work from the original size-2 jobs uncompleted at time 2, when an additional $2m$ units of work are released. Therefore, the on-line algorithm must complete $2m + 5m/18 - m\epsilon/2$ units of work by time 4. There are $2m + 2m\epsilon$ units available on the $m$ speed-$(1+\epsilon)$ machines from time 2 to time 4. Therefore we have $2m + 2m\epsilon \geq 2m + 5m/18 - m\epsilon/2$, which implies $\epsilon \geq 1/9$.

In scenario B, from time 0 to 1, the on-line algorithm completes at most $m + m\epsilon - 2m/9$ units of work on the size-1 jobs. Therefore at time 1, there are at least $2m/9 - m\epsilon$ units of original size-1 work remaining when the additional $m$ units of size-1 work are released. These $m + 2m/9 - m\epsilon$ units must be completed by time 2 with the $m + m\epsilon$ resources available. Therefore $m + m\epsilon \geq m + 2m/9 - m\epsilon$, which implies $\epsilon \geq 1/9$.

To extend this result to 1/5, we use the infinite family of input instances $I(m, i)$ for $i \geq 0$ defined as follows. At time $2j$ for $0 \leq j \leq i$, $m$ jobs are released with execution times 1 and deadlines $2j+2$, and $m/2$ jobs are released with execution times 2 and deadlines $2j+4$. At time $2i+1$, $m$ jobs are released with execution times 1 and deadlines $2i+2$. A careful analysis of this set of input instances reveals that an on-line algorithm must complete all $3m/2$ jobs released at time $2j$ by time $2j+2$ while leaving enough spare capacity from time $2j$ to $2j+1$ to prepare for the possibility that $m$ jobs of size 1 and deadlines of time $2j+2$ will be released at time $2j+1$.

To extend the 1/9 lower bound for real-time scheduling to a 1/21 lower bound for average flow time, we utilize a similar adversary strategy, but the analysis is more complex. □

We note Lemma 2.3 has two other implications. First, Theorem 2.1 can be generalized to show that for $1 \leq \alpha \leq 2 - \frac{1}{m}$, SRPT is an $\alpha$-speed $\frac{2-1/m}{\alpha}$-approximation algorithm for preemptively minimizing the sum of completion times. When $\alpha = 1$, SRPT is a $(2 - \frac{1}{m})$-approximation algorithm which improves slightly a bound of 2 from [20]. Second, Lemma 2.3 can be viewed as a generalization of Graham's proof that List Scheduling is a $(2 - \frac{1}{m})$-approximation algorithm for minimizing the makespan [7]; it is an $\alpha$-speed $(2 - \frac{1}{m})/\alpha$-approximation algorithm.

## 3 On-line preemptive scheduling using more machines

Another natural way in which to augment resources is to add extra machines of the same speed rather than increasing machine speed. In the preemptive setting, it is easy to verify that augmenting with extra machines of the same speed is not superior to augmenting with faster machines and may in fact be worse. More precisely, in the preemptive setting, any $c$-machine algorithm can be simulated by a $c$-speed algorithm by doing a unit of work from each machine during each time unit, while the reverse is not true. In particular, the natural analogue to Lemma 2.3 for extra machines is not true; in particular, it is not true that there exists some constant $c$ such that a busy algorithm given $cm$ machines will always do as much work by any time $t$ as an optimal algorithm given only $m$ machines on any input instance. We exploit this difference to show that LLF is not a $c$-machine algorithm for any constant $c$ and that EDF is not an $o(\Delta)$-machine algorithm. We then show that LLF is an $O(\log \Delta)$-machine algorithm and is thus significantly better than EDF for online preemptive real-time scheduling in this model. Finally, we show that no $(1+\epsilon)$-machine algorithm

exists for this problem for $\epsilon < 1/4$.

To show that LLF is not a $c$-machine algorithm for any $m \geq 2$, we require the following lemma showing that LLF cannot offer a guarantee similar to that of Lemma 2.3. That is, we will show there exist input instances $I$ and times $t$ where an algorithm (particularly the optimal one), using only $m$ machines, completes more work by time $t$ when run on $I$ (in particular, the algorithm finishes all jobs in $I$ by time $t$) than LLF using $cm$ machines for any constant $c$ (in particular, LLF has not completed all jobs in $I$ by time $t$).

**Definition 3.1** *Define the job set $I(t, c, y, m)$ as follows. There are $\frac{m}{2}$ type-A jobs with length $2cy$, release times $t$, and deadlines $t + (2c+1)y$ (and thus laxities of $y$). Meanwhile, there are $cm$ "delay" jobs with release times $t + 2ci$, length 1, and deadlines $t + 2c(i+1)$ (and thus laxities of $2c-1$) for $0 \leq i \leq y-1$ (and thus $ycm$ delay jobs altogether).*

**Lemma 3.1** *For all $m \geq 2$, integers $c \geq 1$, $y \geq 2c$, and $t \geq 0$, job set $I(t, c, y, m)$ can be legally scheduled by time $2cy$ on $m$ machines, but if LLF is run on $I(t, c, y, m)$ with $cm$ machines, at time $2cy + t$, the $\frac{m}{2}$ type-A jobs of $I(t, c, y, m)$ will each have $y - 2c + 1$ remaining processing time and $2c - 1$ laxity.*

*Proof:* For $0 \leq +i \leq y - 2c$, at time $t + i$, the laxity of each of the type-A jobs is at least $2c$, so LLF will schedule all the delay jobs at their release dates. Thus the type-A jobs will not be executed at time $t + 2ci$ for $0 \leq i \leq y - 2c$. At time $t + 2cy$, the $\frac{m}{2}$ type-A jobs will still each require $y - 2c + 1$ units of processing and will have laxities $2c - 1$. However, all jobs can be completed legally by time $t + 2cy$ on $m$ machines by executing the type-A jobs on $\frac{m}{2}$ machines from time $t$ to $t + 2cy$, and packing the delay jobs greedily on the remaining $\frac{m}{2}$ machines. □

**Theorem 3.2** *LLF is not a $c$-machine algorithm for scheduling with deadlines for any constant $c$.*

*Proof Sketch:* Let $y_1 = 4(2c-1)(2c+1)^{4c-1}$ (the key property is that $y_1 > 2(2c-1)(2c+1)^{4c-1}$). We recursively define $y_i = y_{i-1}/(2c+1)$ for $2 \leq i \leq 4c$ (note $y_{i-1} = (2c+1)y_i$ and $y_1 = (2c+1)^{4c-1}y_{4c}$). Let $t_i = 2c\sum_j^i y_i$ for $0 \leq i \leq 4c$ (note $t_0 = 0$). Consider the input instance $I$ consisting of $\cup_{i=1}^{4c} I(t_{i-1}, c, y_i, m)$.

The key to the proof is the observation that at time $t_i$ for $1 \leq i \leq 4c$, LLF will still have $\frac{mi}{2}$ jobs with at least $y_i - 2c + 1$ required units of processing remaining and laxities of at most $2c - 1$ whereas the optimal $m$ machine algorithm finishes all jobs by their deadlines. We omit the inductive proof of this observation in this writeup.

Thus, at time $t_{4c} = 2c\sum_{j=1}^{4c} y_j$, there will be $2cm$ jobs with $y_{4c} - 2c + 1$ required units of processing remaining and laxities $2c - 1$ for LLF. If $y_{4c} > 2(2c-1)$, LLF will not be able to legally schedule this instance. Working backwards, we see that since we defined $y_1 = 4(2c-1)(2c+1)^{4c-1}$, $y_{4c} = 4(2c-1) > 2(2c-1)$, and the lemma follows. □

Note the lower bound instance has extremely large $\Delta$. We can show this is necessary. We can also give an upper bound on the performance of LLF.

**Theorem 3.3** *LLF is an $O(\log \Delta)$-machine algorithm for scheduling with deadlines.*

In contrast, we can show that EDF cannot offer a similar guarantee.

**Theorem 3.4** *EDF is not a $(\lfloor \Delta \frac{m-1}{m} \rfloor)$-machine algorithm for scheduling with deadlines.*

*Proof:* For all $m \geq 2$ and all $\Delta \geq 1$, we define an infinite set of input instances $I(m, \Delta)$ such that $I(m, \Delta)$ can be scheduled by a static algorithm on $m$ machines but cannot be scheduled by EDF on $cm$ machines where $c = \lfloor \Delta \frac{m-1}{m} \rfloor$.

The input instance $I(m, \Delta)$ consists of $cm$ jobs with deadlines of $2\Delta$ and execution times of 2 and one job with a deadline of $2\Delta + 1$ and an execution time of $2\Delta$ (i.e. this job must be executed by time 1 in order to complete by its deadline). All the jobs are released at time 0.

Clearly, EDF will fail to legally schedule the input instance $I(m, \Delta)$ on $cm$ machines because it will initially devote all $cm$ machines to the scheduling of the short jobs with length 2 and will not schedule the long job until time 2 at which point it can no longer be completed by its deadline.

The input instance $I(m, \Delta)$, however, is clearly schedulable on $m$ machines as we can put the one job with deadline $2\Delta + 1$ on a machine at time 0 and devote the remaining $m - 1$ machines to completing the remaining $cm$ jobs of length 2 by their deadlines. □

Finally, we also can show that no algorithm is a $(1+\epsilon)$-machine algorithm for this problem. In the theorem below, we assume that $\epsilon m$ is an integer.

**Theorem 3.5** *There is no on-line $(1+\epsilon)$-machine algorithm for scheduling with deadlines on $m$ machines, $m \geq 1$ and even, for any $\epsilon < \frac{1}{4}$.*

## 4 On-line preemptive weighted flow time

In this section we give an on-line extra-resource algorithm for the preemptive scheduling of one machine to minimize average *weighted* flow time. Our approach will build upon the work of Hall et. al. [9] who recently showed that a variety of linear programming formulations can be used to give approximation algorithms to minimize the $\sum w_j C_j$ objective. As we observed earlier, while an optimal algorithm for $\sum w_j C_j$ is also an optimal algorithm for $\sum w_j F_j$, a $\rho$-approximation algorithm for $\sum w_j C_j$ in no way is guaranteed to be a $\rho$-approximation algorithm for $\sum w_j F_j$. No non-trivial polynomial-time algorithm is known for this problem.

Let $N$ denote denote the entire set of jobs $\{1, \ldots, n\}$. For any subset $S \subseteq N$, we use the notation $p(S) = \sum_{j \in S} p_j$, and $r_{\min}(S) = \min_{j \in S} r_j$. For any schedule of $N$, let $C_j$ denote the completion time of job $j$. The following lemma places constraints on legal values for $C_j$.

**Lemma 4.1** *[21] Given a preemptive schedule for instance $N$ on one machine, let $C_1, \ldots, C_n$ denote the job completion times in this schedule. Then the $C_j$ satisfy the inequalities*

$$\sum_{j \in S} p_j C_j \geq r_{\min}(S)p(S) + \frac{1}{2}\left(p(S)^2 + p^2(S)\right) \quad \text{for each } S \subseteq N. \tag{3}$$

We now form the following linear program $R$:

$$\text{Minimize } \sum_{j=1}^{n} w_j C_j \text{ subject to (3) and the}$$

$$n \text{ constraints } C_j \geq r_j + p_j \text{ for } j = 1, \ldots, n.$$

While $R$ has an exponential number of constraints, Goemans has shown that $R$ is a linear transformation of a supermodular polyhedron. This means we may obtain an optimal solution to $R$ in polynomial time by applying the greedy algorithm[4]; in fact, Goemans shows we can solve it in $O(n \log n)$ time.

Hall et. al. [9] show that the solution to this relaxation can be used to produce a 2-approximation algorithm for $\sum_j w_j C_j$; this has been improved to a 1.466-approximation algorithm [6]. We show here that a similar approach yields a 2-speed algorithm for $\sum_j w_j F_j$. We first present an off-line algorithm and then convert it into an on-line algorithm.

Compute the optimal linear programming solution to $R$ given by (3) and denote this solution $\overline{C}_1, \ldots, \overline{C}_n$. We use this solution to induce a set of priorities on the $n$ jobs; specifically, job $i$ has priority over job $j$ if $\overline{C}_i < \overline{C}_j$. At any time, our algorithm schedules *one-half* of the available job with highest priority; clearly a machine of speed 2 could schedule the entire job in this time. We call this algorithm Preemptively-Schedule-Halves-by-$\overline{C}_j$.

**Lemma 4.2** *Let $\overline{C}_1 \leq \cdots \leq \overline{C}_n$ be an optimal solution to $R$, and let $\widetilde{C}_1, \ldots, \widetilde{C}_n$ denote the completion times found by* Preemptively-Schedule-Halves-by-$\overline{C}_j$. *Then, for $j = 1, \ldots, n$, $\widetilde{C}_j \leq \overline{C}_j$.*

*Proof Sketch:* Consider the schedule formed on the first $j$ jobs. Let $t$ be the latest time before $\widetilde{C}_j$ at which the machine is idle. (If no such time exists, $t = 0$). Let $S$ denote the set of jobs that are at least partially processed in the interval $[t, \widetilde{C}_j]$, in the partial schedule. We observe that all jobs in $S$ were released after time $t$ because if not, they would be running at time $t$. Therefore, $t = r_{\min}(S)$, and since there is no idle time between $t$ and $\widetilde{C}_j$, $\widetilde{C}_j \leq r_{\min}(S) + \frac{1}{2}p(S)$.

Now consider inequality (3) on set $S$. Since we are only considering the partial schedule of the first $S$ jobs, for all $k \in S$, $\overline{C}_k \leq \overline{C}_j$. Plugging this into (3) gives that $\overline{C}_j p(S) \geq \sum_{k \in S} p_k \overline{C}_k \geq r_{\min}(S)p(S) + \frac{1}{2}p(S)^2$, or $\overline{C}_j \geq r_{\min}(S) + \frac{1}{2}p(S)$. Thus $\widetilde{C}_j \leq \overline{C}_j$, as we wished to show. □

We convert this into an on-line algorithm as follows. At any time, we have a set of released jobs. We form the linear program for these jobs, solve it to compute priorities on jobs, and greedily schedule according to these priorities until a new job arrives. This on-line algorithm performs identically to the off-line algorithm because the relative priorities of jobs that arrive before any time $t$ are unaffected by the arrival of jobs after time $t$ [5].

**Theorem 4.3** Preemptively-Schedule-Halves-by-$\overline{C}_j$ *is an on-line 2-speed algorithm for scheduling preemptively to minimize $\sum w_j F_j$.*

## 5 On-line nonpreemptive scheduling

In this section, we consider the significantly more difficult problem of nonpreemptive scheduling. The main problem we address is minimizing total weighted flow time, though we do consider others as well. We first give algorithms that show how to schedule nonpreemptively using more resources. Our approach is to group jobs into groups of similar-sized jobs and use a greedy algorithm to schedule similar sized jobs on the same machines. We then give a lower bound on nonpreemptive scheduling and some comments relating the upper and lower bounds.

### 5.1 Algorithms

In this section, we will use *flow-time problem* to refer to the problem of nonpreemptively minimizing the total weighted flow time on $m$ identical parallel machines. We will compare several different algorithms and the schedules that they produce. Given an input $I$, we will denote

- $O(I)$ - the optimal schedule on $m$ machines,
- $G(I)$ - the greedy algorithm on $m$ machines. When a machine is idle, any available job is run,
- $Gp(I)$ - a modified greedy algorithm on $m$ machines. When a machine becomes idle, at the next time which is an integer multiple of $p$, the available job with largest weight is scheduled.

We will slightly overload the notation, and use $O(I), G(I)$, $Gp(I)$ to refer both to the algorithms and to the schedules that they produce. We use the notation $F_j^{X(I)}$ to denote the flow time of job $j$ when algorithm $X$ is run on input $I$, and $S_j^{X(I)}$ to denote the starting time of job $j$ when algorithm $X$ is run on input $I$.

We first focus on input instances consisting of similar-sized jobs. We will then discuss how to generalize this algorithm to schedule arbitrary-sized jobs.

**Similar Sized Jobs:** We begin with the case when all $p_j$ are the same and all $w_j = 1$, and observe that in this case, the greedy algorithm $G$ is optimal since if a job is held up for the release of another job of the same size, the two can be swapped, improving the flow time.

**Lemma 5.1** *Let $p$ be a positive integer. Let $I$ be a flow-time problem with $w_j = 1$ and $p_j = p$ for all jobs $j$. Then $\sum_j F_j^{G(I)} = \sum_j F_j^{O(I)}$.*

We now extend this to the case when all $p_j$ are within a factor of 2 of each other.

**Lemma 5.2** *Let $p$ be a positive integer. Let $I$ be a flow time problem on $m$ machines with $w_j = 1$ and $p \leq p_j \leq 2p$ for all jobs $j$. There is an on-line algorithm $U$, run on $2m$ machines, that given input $I$ produces a schedule with $\sum_j F_j^{U(I)} \leq \sum_j F_j^{O(I)}$. Further, there exists an optimal schedule $O$ for which $S_j^{U(I)} \leq S_j^{O(I)}$ for all jobs $j$.*

*Proof Sketch:* Let $I'$ denote a modified instance of $I$, with all $p_j = p$. By Lemma 5.1, if we run the greedy algorithm $G$ on $I'$ we get that $\sum_j F_j^{G(I')} = \sum F_j^{O(I')}$ and hence there exists an optimal schedule (namely $G$) in which for all $j$,

$$S_j^{G(I')} = S_j^{O(I')} \tag{4}$$

Since $I'$ is formed from $I$ by decreasing processing times, we can show that there exist optimal schedules for $I'$ and $I$ for which $S_j^{O(I')} \leq S_j^{O(I)}$ and so, combining with(4) we have

$$S_j^{G(I')} \leq S_j^{O(I)} \tag{5}$$

and also that $\sum F_j^{G(I')} \leq \sum F_j^{O(I)}$. In other words, $G$, run on $I'$ does "better" than the optimal schedule for $I$.

Now, we form the on-line $2m$ machine algorithm $U$ on $I$ by simulating $G$ on $I'$. We devote 2 machines in $U$ for the

jobs scheduled on each machine in $G(I')$, and alternate placing the jobs from a machine in $G(I')$ on the corresponding two machines in $U$. Even though, in converting from $I'$ back to $I$ processing times can double, we are guaranteed that, because of the alternation, we will be able to schedule each job at exactly its start time in $G$, that is, $S_j^{U(I)} = S_j^{G(I')}$ for all $j$. Combing with (5), we conclude that $S_j^{U(I)} \leq S_j^{O(I)}$ for all $j$, and thus $\sum_j F_j^{U(I)} \leq \sum_j F_j^{O(I)}$. ■

We now consider input instances in which the jobs have different weights. We again begin by focusing on the case when all jobs have the same processing time $p$. In this case greedy is not an optimal algorithm, but we can show that a slightly modified version, in which jobs start at times which are integral multiples of $p$, comes close to optimal. The following lemma shows that considering such schedules does not increase the objective function much.

**Lemma 5.3** *Let $p$ be a positive integer. Let $I$ be a flow time problem with $p_j = p$ for all jobs $j$. Then*

$$\sum w_j F_j^{Gp(I)} \leq \sum w_j (F_j^{O(I)} + p).$$

*Proof Sketch:* We first constrain ourselves to schedules in which each job is required to start at a time which is an integral multiple of $p$, and we let $Op(I)$ be the optimal schedule in this setting. We first show that $\sum w_j F_j^{Op(I)} \leq \sum w_j (F_j^{O(I)} + p)$. To prove this, take optimal schedule $O$, and on each machine, move each job later so that it begins at an integral multiple of $p$. Clearly we still have a valid schedule, and the completion time of each job has increased by no more than $p$. Thus we have a schedule in which each job begins at a time which is an integral multiple of $p$ and has flow time no more than $\sum w_j (F_j^O + p)$. Thus $Op$, the optimal such schedule has flow time no greater than this. We can then show that $Gp$ is indeed an optimal schedule in the setting in which each job is required to start at a time which is an integral multiple of $p$. We omit the details. ■

Since $F_j^{O(I)} \geq p$ for all $j$, $Gp$ finds a schedule for weighted flow time with all $p_j = p$ with flow time within a factor of 2 of optimal. We now use a proof similar to that of Lemma 5.2 to bound the performance of an on-line $2m$ machine version of $Gp$, on the input instances where $p \leq p_j \leq 2p$ for some $p$.

**Lemma 5.4** *Let $p$ be a positive integer. Let $I$ be a flow time problem on $m$ with $p \leq p_j \leq 2p$. There is an on-line algorithm $U$, that given $I$ and $2m$ machines, produces a schedule $U$ with $\sum w_j F_j^{U(I)} \leq 2 \sum w_j F_j^{O(I)}$.*

*Proof Sketch:* Form $I'$ from $I$ by rounding the processing times down to $p$. We know by Lemma 5.3 that if we run $Gp$ on $I'$, that $\sum w_j F_j^{Gp(I')} \leq \sum w_j (F_j^{O(I')} + p)$. On-line algorithm $U$ is formed exactly as in the proof of Lemma 5.2: we assign two machines to each one and then simulate $Gp$ on $I'$ and use this to obtain starting times in the on-line algorithm. Thus we have that $S_j^{U(I)} = S_j^{Gp(I')}$. We also know that in $I'$, all jobs have processing time exactly $p$, and so $S_j^{Gp(I')} = F_j^{Gp(I')} - p$. Combining these bounds, we get

$$\begin{aligned}
\sum w_j F_j^{U(I)} &= \sum w_j (S_j^{U(I)} + p_j) \\
&= \sum w_j (S_j^{Gp(I')} + p_j) \\
&= \sum w_j (F_j^{Gp(I')} - p + p_j) \\
&\leq \sum w_j (F_j^{O(I')} + p_j) \\
&\leq \sum w_j F_j^{O(I)} + \sum w_j p_j \\
&\leq 2 \sum w_j F_j^{O(I)}
\end{aligned}$$

■

The results of this section also hold with speed-2 machines instead of doubling the number of machines.

**General Algorithms:** We now give algorithms for jobs with arbitrary-sized-processing times. We split the jobs into groups of similarly sized jobs, and put each group on its own machine. There will be a logarithmic number of groups, and for each group we can use the algorithms for when the processing times are all between $p$ and $2p$.

Let $p_{\min} = \min_j p_j$ and $p_{\max} = \max_j p_j$. Recall that $\Delta$ is defined as $\max p_j / \min p_j$, the ratio between the minimum and maximum processing times.

**Theorem 5.5** *There is an on-line $2\lceil \log \Delta \rceil$-machine algorithm for minimizing total flow time on $m$ machines, an on-line $O(\log n)$-machines $(1 + o(1))$-approximation algorithm for minimizing total flow time, and an on-line $O(\log n)$-machine $(1+o(1))$-speed algorithm for minimizing total flow time.*

*Proof Sketch:* We divide the jobs into $\lceil \log \Delta \rceil$ groups, where the $i$th group contains all the jobs with $2^{i-1} p_{\min} \leq p_j \leq 2^i p_{\min}$. Within each group, processing times differ by at most a factor of 2 and hence Lemma 5.2 can be applied. When $\Delta$ is large, we can replace $\lceil \log \Delta \rceil$ by $3 \log n + 1$ while sacrificing a small factor in the total flow time. To do so, we use only the $3 \log n$ largest groups and one additional group for all the remaining small jobs. More precisely, there are groups for jobs in the ranges $(p_{\max}, p_{\max}/2)$,$(p_{\max}/2, p_{\max}/4)$,...,$(2p_{\max}/n^3, p_{\max}/n^3)$ and one group for all the remaining jobs. We schedule all the groups except for the last optimally using Lemma 5.2. For the final group, the total amount of processing is no more than $n(p_{\max}/n^3) = p_{\max}/n^2$. Therefore, each job has flow time no more than $p_{\max}/n^2$, and the total flow time is no more than $n(p_{\max}/n^2) = p_{\max}/n$. Since the total flow time for the original input is clearly at least $p_{\max}$, this multiplies the total flow time by a $1 + o(1)$ factor. If we instead use speed-$(\frac{n}{n-1})$ machines, since the jobs are scheduled nonpreemptively, the flowtime of the largest job is reduced by a factor of $\frac{n-1}{n}$, which by itself is enough to absorb the total flow time of the small jobs. ■

For weighted flow time, we divide the processing times into groups differing by factors of two and apply Lemma 5.4 to obtain the following theorem:

**Theorem 5.6** *There is an on-line $2\lceil \log \Delta \rceil$-machine 2-approximation algorithm for minimizing weighted flow time on $m$ machines.*

Unfortunately, we do not know how to convert this into an algorithm using $O(\log n)$ machines, as it might be the case that all the weight is on the jobs with small processing times.

We can also apply these techniques to the problem of nonpreemptive scheduling with deadlines. We omit the details in this extended abstract.

**Theorem 5.7** *There is a 4-speed $O(\log \Delta)$-machine algorithm for scheduling to nonpreemptively meet deadlines.*

### 5.2 Lower Bounds

We now present a theorem that indicates that it may be difficult to improve upon our results. A fundamental lower bound for nonpreemptive average flow time is the optimum for the corresponding preemptive problem; in this section we show that any extra-machine algorithm whose analysis is based on a comparison to this lower bound must do poorly. Specifically, we give a lower bound on the power of additional machines when we are nonpreemptively scheduling and wish to achieve the same flow time as the optimal preemptive schedule. Obviously, there are input instances where the optimal nonpreemptive flow time on $c$ machines can be significantly better than the optimal preemptive flow time on a single machine. We now show the surprising result that there are input instances where the optimal nonpreemptive flow time on $c$ machines is significantly worse than the optimal preemptive flow time on a single machine for any natural number $c$. This generalizes the result of [13], who show that there exist input instances where the optimal nonpreemptive flow time on a single machine may be $\Theta(n^{1/2})$ times greater than the optimal preemptive flow time on a single machine.

**Theorem 5.8** *There exists a family of input instances $I(c, N)$ with $\Theta(N)$ jobs such that the optimal nonpreemptive flow time for input instance $I(c, N)$ on $c$ machines is $\Omega(N^{\frac{1}{2^{c+1}-2}})$ times greater than the optimal preemptive flow time for input instance $I(c, N)$ on one machine, for large enough $N$.*

*Proof:* Given $N$ and constant $c$, define $n = N^{\frac{1}{2^{c+1}-2}}$. We will construct instance $I(c, N)$ with between $N$ and $2N$ jobs such that the optimal preemptive flow time for $I(c, N)$ on one machine is $\Theta(N)$ while the optimal nonpreemptive flow time for $I(c, N)$ on $c$ machines is $\Omega(Nn)$.

The instance $I(c, N)$ is constructed using $c + 1$ different types of jobs which we number from 0 to $c$. For each job type $i$, $0 \leq i \leq c$, there are $num(i) = n^{2^{i+1}-2}$ different jobs, each of length $len(i) = \frac{N}{num(i)}$. Jobs of type $i$ arrive every $n\ len(i)$ time units starting at time 0 until all of the $num(i)$ type $i$ jobs have arrived. A key property of instance $I(c, N)$ is that during any time interval of length $len(i)$ between time 0 and time $Nn$, $\frac{Nn}{len(i)}$ type $i + 1$ jobs arrive.

We first observe that the optimal preemptive flow time for $I(c, N)$ on one machine is lower bounded by the sum of the lengths of all jobs which is $(c + 1)N$. We now show that the optimal preemptive flow time is upper bounded by $2(c + 1)N$. Consider the algorithm in which we greedily schedule each job type in turn, starting with the jobs of type $c$, and going down. A simple induction proof shows that this algorithm delays no job for more than its execution time and the $2(c + 1)N$ upper bound on the optimal preemptive flow time follows.

We now show that the optimal nonpreemptive flow time for $I(c, N)$ is $\Omega(Nn)$. We do this by showing, for each $k$, $1 \leq k \leq c$, that there must be some time between time 0 and time $Nn$ during which $k$ jobs, one type $i$ job for $0 \leq i < k$, must be run simultaneously for $len(k - 1)$ time or else the nonpreemptive flow time is $\Omega(Nn)$.

We prove this by induction on $k$. For the base case $k = 1$, this means the one type 0 job must complete execution before time $Nn$ or else the flow time of the schedule is $\Omega(Nn)$. This is clearly true since if the one type 0 job which is released at time 0 does not complete before time $Nn$, its flow time is $Nn$.

For the inductive case, assume we have shown, for $1 \leq k < c$, that there exists a time interval from time 0 to time $Nn$ during which $k$ jobs, one type $i$ job for $0 \leq i < k$, must be run simultaneously for $len(k - 1)$ time or else the nonpreemptive flow time is $\Omega(Nn)$ We now show this must hold as well for $k = c$.

Consider the time interval of length $len(k - 1)$ during which $k$ jobs are running simultaneously. From the key property we described earlier, we know that $\frac{Nn}{len(k-1)}$ type $k$ jobs are released during this time interval. If none of these jobs complete before the end of this time interval, the flow time of these jobs is $\Theta(Nn)$. Therefore, one of these jobs must complete before the end of this time interval. Since these jobs arrived after the beginning of the interval, the induction hypothesis holds for $k + 1$.

Therefore, we know either the optimal nonpreemptive schedule has flow time $Nn$ or there must be some time $len(c - 1)$ time interval prior to time $Nn$ when $c$ jobs execute simultaneously on the $c$ machines. Therefore, the $\frac{Nn}{len(c)}$ type $c$ jobs that arrive during this time interval cannot begin execution until this interval ends. This means these jobs have a flow time of at best $\Theta(Nn)$. Thus, the optimal nonpreemptive flow time for $I(c, N)$ is $\Omega(Nn)$. □

The proof of Theorem 5.8 can be modified to show a polynomial-size gap between the preemptive flow time on one machine and the nonpreemptive flow time on any constant number of machines even if the machines are speed-2. Thus the logarithmic-machine speed-$(1 + o(1))$ algorithm of Theorem 5.5 produces a nonpreemptive schedule with flowtime potentially polynomially better than the optimal speed-2 schedule for any constant number of machines.

In the nonpreemptive setting, additional speed may not be as powerful as additional machines. We can give an example in which adding a single extra machine is a significantly better approach as we can show that there exists a set of jobs for which the optimal preemptive average flow time can be achieved on 2 speed-1 machines, but cannot be achieved on a single speed-$c$ machine for any $c < (n/2)^{1/4}$.

## 6 Translating faster machine results to stretch results

In this section we show how algorithms for minimizing average flow time in the faster-machine model translate to algorithms on machines of the same speed for stretched input instances.

**Theorem 6.1** *If $A$ is an $s$-speed $\rho$-approximation algorithm for minimizing average flow time in any model (preemptive or nonpreemptive, clairvoyant or nonclairvoyant, on-line or offline), then there exists an algorithm $A'$ which is an $s$-stretch $\rho s$-approximation algorithm for minimizing average flow time.*

*Proof:* Remember that for any input instance $I$, $I^s$ is the identical input instance except job $J_i$ has release time $r_i s$ for $1 \leq i \leq n$. The basic idea is that at any time $ts$, $A'$ behaves exactly as $A$ did at time $t$. Because of the above relationship between $I$ and $I^s$, $A'$ is well defined.

Let $C_j$ and $F_j$ denote the completion time and flow time, respectively, of job $J_j$ when $A$ schedules input instance $I$, and $C'_j$ and $F'_j$ denote the completion time and flow time, respectively, of job $J_j$ when $A'$ schedules input instance $I^s$.

It is not hard to see that $C'_j = sC_j$ 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. □

Note the theorem holds for any scheduling model and leads to the following series of results.

**Corollary 6.2** *The Balance algorithm [11] is an s-stretch $\left(s + \frac{s}{s-1}\right)$-approximation algorithm for minimizing average flow time on a single processor when execution times of jobs are unknown until they complete. In particular, the value $s = 2$ leads to a minimum approximation ratio of 4.*

**Corollary 6.3** *SRPT is a $(2 - 1/m)$-stretch $(2 - 1/m)$-approximation algorithm for minimizing average flow time on multiple processors.*

**Corollary 6.4** *The algorithm* Preemptively-Schedule-Halves-by-$\overline{C}_j$ *is a 2-stretch 2-approximation algorithm for minimizing average weighted flow time on a single processor.*

Note faster machine results for real-time scheduling extend to stretched input results for real-time scheduling if and only if the deadlines of jobs are multiplied by a factor of $s$ as well. Also, while extra machine results translate to stretched input results in a preemptive environment, extra machine results do not seem to translate to stretched input results in a nonpreemptive environment. Thus, our results in Section 5 do not translate into stretched input results for nonpreemptive scheduling.

**Acknowledgments** We are grateful to Bala Kalyanasundaram, Stavros Kolliopoulos, Stefano Leonardi, Rajeev Motwani, Steven Phillips, Kirk Pruhs, and David Shmoys for useful discussions.

## References

[1] S. Chakrabarti, C. A. Phillips, A. S. Schulz, D. B. Shmoys, C. Stein, and J. Wein. Improved scheduling algorithms for minsum criteria. In F. Meyer auf der Heide and B. Monien, editors, *Automata, Languages and Programming*, number 1099 in Lecture Notes in Computer Science. Springer, Berlin, 1996. Proceedings of the 23rd International Colloquium (ICALP'96).

[2] M. Dertouzos. Control robotics: the procedural control of physical processes. In *Proc. IFIF Congress*, pages 807–813, 1974.

[3] M. Dertouzos and A. Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15:1497–1506, 1989.

[4] M. Goemans. A supermodular relaxation for scheduling with release dates. In *Proceedings of the 5th Conference on Integer Programming and Combinatorial Optimization*, pages 288–300, June 1996. Published as Lecture Notes in Computer Science 1084, Springer-Verlag.

[5] M. Goemans. Improved approximation algorithms for scheduling with release dates. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms*, pages 591–598, 1997.

[6] M. Goemans, J. Wein, and D. P. Williamson. Randomized algorithms for improved preemptive scheduling. Working paper, 1996.

[7] R.L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[8] D. Gusfield. Bounds for naive multiple machine scheduling with release times and deadlines. *Journal of Algorithms*, 5:1–6, 1984.

[9] L. A. Hall, A.S. Schulz, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time: Offline and on-line algorithms. To appear in Math of OR, 1996.

[10] J.A. Hoogeveen and A.P.A. Vestjens. Optimal on-line algorithms for single-machine scheduling. In *Proceedings of the 5th Conference on Integer Programming and Combinatorial Optimization*, pages 404–414, 1996. Published as Lecture Notes in Computer Science 1084, Springer-Verlag.

[11] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 214–221, 1995.

[12] Bala Kalyanasundaram and K. Pruhs, 1996. Personal Communication.

[13] H. Kellerer, T. Tautenhahn, and G. J. Woeginger. Approximability and nonapproximability results for minimizing total flow time on a single machine. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, pages 418–426, May 1996.

[14] G. Koren, D. Shasha, and S.-C. Huang. Moca: A multiprocessor on-line competitive algorithm for real-time system scheduling. In *Proc. 14th Real-Time Systems Symposium*, pages 172–181, 1993.

[15] J. Labetoulle, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinooy Kan. Preemptive scheduling of uniform machines subject to release dates. In W.R. Pulleyblank, editor, *Progress in Combinatorial Optimization*, pages 245–261. Academic Press, 1984.

[16] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, 1997. Appears in this Conference Proceedings.

[17] J. Leung. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica*, 4:209–219, 1989.

[18] A. Mok. Task scheduling in the control robotics environment. Technical Report TM-77, Laboratory of Computer Science, Massachusetts Institute of Technology, 1976.

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

[20] C. Phillips, C. Stein, and J. Wein. Minimizing average completion time in the presence of release dates. To appear in Math Programming, 1995.

[21] M. Queyranne and A.S. Schulz. Polyhedral approaches to machine scheduling. Technical Report Technical Report 474/1995, Technical University of Berlin, 1994.

[22] S. Sahni and Y. Cho. Nearly on line scheduling of a uniform processor system with release times. *SIAM Journal on Computing*, 8:275–285, 1979.

[23] David B. Shmoys, Joel Wein, and David P. Williamson. Scheduling parallel machines on-line. *SIAM Journal on Computing*, 24(6):1313–1331, December 1995.