Optimal Time-Critical Scheduling Via Resource Augmentation*

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 offline 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 offline 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.

*A preliminary version of this work appeared in the proceedings of the 1997 ACM Symposium on Theory of Computing.

†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@cse.msu.edu. Department of Computer Science and Engineering, 3115 Engineering Building, Michigan State University, East Lansing, MI 48824. Research partially supported by NSF CAREER grant CCR-9701679, a grant from IBM, and an MSU research initiation grant. Some preliminary work was done while this 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.

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 [5]. 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 ([**?**, 19], summarized in Section 1.3.2) show that no on-line algorithm can guarantee a constant approximation ratio. The net result is that traditional worst-case analysis techniques have failed to provide convincing evidence that any algorithm can achieve good performance for any of these problems. Furthermore, for the hard-real-time scheduling problem and the nonpreemptive scheduling to minimize flow time problem, traditional worst-case analysis techniques have failed to differentiate 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 [12] (for one-processor scheduling) of comparing the performance of an on-line algorithm to the performance of an optimal offline 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. Finally, for the hard-real-time scheduling problem, our extra machine results differentiate the performance of the *least-laxity-first* algorithm from the performance of the *earliest-deadline-first* algorithm.

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

[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 Δ is the ratio between the weights of the most important and least important jobs in the input instance [**?**].

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 it will perform well in practice, especially if the instances on which it performs very poorly under traditional worst-case analysis have a special pathological structure. 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.

1.1 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 $\triangle(I)$, or simply $\triangle$ 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.

1.2 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. [9, 10, 11, 25, 26], 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 ρV using m speed-s machines. A *w-machine ρ-approximation algorithm* finds a solution of value ρV using wm machines. An l-

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

Figure 1: Summary of main algorithms and hardness results. The notation $x \leq s \leq y$ means the problem can be solved with speed-y machines, but cannot be solved optimally with speed-$(x - \epsilon)$ machines for any $\epsilon > 0$. 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_{all} to denote all constants. $\triangle$ is the ratio of the longest length to the shortest length.

stretch ρ-approximation finds a solution to I^l of value ρV 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 "ρ-approximation" when this is true.

1.3 Our Results

We now discuss our results, which are summarized in Figure 1.

1.3.1 Preemptive Problems

We first study on-line preemptive scheduling for both objectives: minimum total flow time and hard real-time scheduling with deadlines. We show that two 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 small ϵ (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) [4] and least-laxity-first (LLF) [21]. 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 [4] and LLF [21] 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 [5]. Nonetheless, EDF and LLF are likely heuristic choices in practice.

In the faster-machine model, we show that EDF and LLF are $(2 - \frac{1}{m})$-speed algorithms for the problem of hard-real-time scheduling and that this result is tight for EDF. 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 times. In the uniprocessor setting ($m = 1$), SRPT is optimal. In the multiprocessor setting ($m \geq 2$), while SRPT can still be considered to be an optimal on-line algorithm within a constant factor, its approximation guarantee is nonconstant; in particular, Leonardi and Raz show that SRPT is a $\Theta(\log \min(n/m, \Delta))$-approximation algorithm, and they provide $\Omega(\log n/m)$ and $\Omega(\log \Delta)$ lower bounds on the competitive ratio of any randomized on-line algorithm [19]. In contrast, we show that SRPT is a $(2 - \frac{1}{m})$-speed algorithm. 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, where the weights reflect job priorities. In contrast to the unweighted problem, scheduling preemptively on a single machine to optimize average weighted flow time is $\mathcal{NP}$-hard [17]. Recently there has been much progress in developing offline approximation algorithms for the related $\mathcal{NP}$-hard problem of minimizing $\sum_j w_j C_j$ [2, 10, **?**, **?**], 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 ρ-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 [10] to develop an (on-line) 2-speed algorithm for this problem.

1.3.2 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 (offline uniprocessor, unweighted) is already a difficult problem with very strong nonapproximability results. In particular, Kellerer, Tautenhahn and Woeginger [**?**] 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 shown that there is no polynomial-time $o(n^{1/3})$-approximation algorithm for the parallel machine case [19]. 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 for the on-line minimization of total weighted flowtime on parallel machines, and we give an $O(\log n)$-machine $(1+o(1))$-approximation algorithm as well as an $O(\log n)$-machine $(1+o(1))$-speed algorithm for the on-line minimization of total flowtime (unweighted) 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, even for unweighted flow time. 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 on the optimal nonpreemptive solution [**?**, **?**]. Let S_p be the schedule with optimal flowtime in the preemptive 1-machine setting for some instance I, and let S_m be the schedule with optimal flowtime in the nonpreemptive m-machine setting. We give a polynomial-size lower bound of $\Theta\left(n^{\frac{1}{2^{m+1}-2}}/s^2\right)$ on the gap between $\sum_j F_j^{S_p}$ and $\sum_j F_j^{S_m}$ for any constant $m \geq 2$, even if the m machines are speed-s for any constant s. Thus the analysis of any $O(1)$-machine s-speed nonpreemptive flow-time algorithm will require a stronger lower bound on the optimal nonpreemptive flow-time than the flow-time of the optimal preemptive 1-machine 1-speed schedule. In contrast to this negative result about $O(1)$-machine algorithms, we give an $O(\log n)$-machine 2-speed algorithm that nonpreemptively achieves the 1-machine 1-speed preemptive lower bound.

1.3.3 Speed and Stretch

We conclude by showing that any s-speed ρ-approximation algorithm is also an s-stretch ρs-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 results for faster machine extend to stretched schedules for all results concerning flow time.

1.4 Related Results

Several papers have applied extra resource analysis to the problem of minimizing flow time in a variety of scheduling environments. Kalyanasundaram and Pruhs were the first to do so in their study of the minimization of preemptive total flow time and best-effort firm-real-time scheduling [12]. For the unweighted uniprocessor setting where the algorithm has no knowledge of p_j until job J_j completes, 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 $\Omega(n^{\frac{1}{3}})$ [22]. This result was improved by Coulston and Berman who show that for $s \geq 2$, the algorithm actually is an s-speed $\frac{2}{s}$-approximation algorithm; that is, for $s > 2$, the algorithm with m speed-s machines outperforms an optimal algorithm for m speed-1 machines [**?**]. Finally, Edmonds has recently shown that Equi-partition, a natural generalization of the round robin algorithm to a parallel processing setting, is a $(2+\epsilon)$-speed $(2+\frac{4}{\epsilon})$-approximation algorithm for $\epsilon > 0$ and an s-speed $\frac{16}{s}$-approximation algorithm for $s \geq 4$ [**?**]. Thus, for $s \geq 16$, Equi-partition can achieve optimal performance. Edmonds also proves many other results about a wide variety of scheduling models with different assumptions about the parallelizability of jobs.

Likewise, several papers have applied extra resource analysis or similar analysis techniques to several real-time scheduling problems. However, nearly all of these problems have allowed jobs to not be completed. For example, Kalyanasundaram and Pruhs consider a problem where jobs have values and the goal is to maximize the sum of the values of completed jobs in a uniprocessor setting [12]. 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. They also consider the unweighted version of this problem and show that either SRPT or a second deterministic algorithm has a constant competitive ratio on every instance for this problem [**?**]. Thus, if an on-line algorithm is given two processors, it will be constant competitive against the optimal offline algorithm on one processor. Other authors have considered best-effort real-time scheduling problems where jobs are assumed to have a minimum percentage laxity [**?**, **?**]. In some ways, this is similar to assuming that the on-line algorithm has faster machines. However, a difference is that both the on-line and offline algorithms benefit from this minimum laxity whereas only the on-line algorithm benefits from extra resources in extra resource analysis.

More recently, Kalyanasundaram and Pruhs have looked at the relationship between migratory and nonmigratory multiprocessor real-time scheduling [14]. In the migratory model, a job may be preempted and then resumed with no penalty on a different processor. In the nonmigratory model, a job must be completely processed by one machine, though it may be preempted. The main result from this paper with respect to our work is that if a set of jobs can be completed by their deadlines on m processors in a migratory environment, then this same set of jobs can be completed by their deadlines on $3m-2$ processors in a nonmigratory environment for $m \geq 1$. Thus, they use extra resources to offset migration whereas we use extra resources to offset knowledge of the future.

Bender, Chakrabarti, and Muthukrishnan have recently produced results dealing with stretch [1]. However, their definition of stretch is different than ours. In particular, given an input instance I and a schedule $S(I)$, they define the stretch of job n in I to be the flow time of job n divided by the length of job n. Their definition of stretch captures the slowdown job n experiences due to the presence of other jobs.

Finally, Tak Wah Lam and Kar Keung To have recently extended the preemptive real-time scheduling results of this paper to allow both extra processors and extra resources [18]. They show that EDF given $1+p$ speed-$(2-(1+p)/(m+p))$ processors can schedule any instance feasible for m speed-1 processors. For $p=0$, this is the bound given in this paper. They show that any algorithm given $1+p$ processors requires these processors to have speed at least $1/(2\sqrt{2+p/m}-2)$ to schedule any instance feasible on m speed-1 machines. When $p=0$, this is slightly stronger (in the 3rd decimal place) than the bound given in this paper. They give an algorithm that schedules all instances feasible for m speed-1 machines using $1+p$ speed-$(2-(2(m-1)+mp)/((m+1)(m-1)+mp))$ processors. This is similar to our bounds for EDF for $p=0$ when m is large. However, for $m=2$ and $p=0$ the speed of $4/3$ improves upon the 1.5 we prove for EDF. For $m=2$ and $p=1$, the new bound beats the lower bound for $p=0$. The paper also gives improved lower bounds for all algorithms that schedule based only on deadline. The new algorithm is optimal for $m=2$ and $p=0$ among such deadline-ordered schedules.

2 On-line preemptive unweighted scheduling

In this section, we consider both scheduling to minimize *unweighted* average flow time and hard-real-time scheduling in a preemptive scheduling environment. We first consider the faster machine setting, and we show that SRPT is a preemptive, $(2-1/m)$-speed algorithm for minimizing average flow time and that EDF and LLF are $(2-1/m)$-speed algorithms for hard-real-time scheduling and that this result is tight for EDF. These results are based upon a fundamental result (Lemma 2.6) which shows a tradeoff between the amount of work done by any busy algorithm by specific times and the speed of the processors it is given.

We next consider the extra machine setting, and we first observe that the natural analogue to Lemma 2.6 does not apply. As a result, we are unable to derive any results for SRPT for minimizing average flow time, and we derive much poorer results for EDF and LLF for hard-real-time scheduling. In particular, we show that both LLF and EDF are not c-machine algorithms for any constant c for the hard-real-time scheduling problem. However, we then show that that LLF is an $O(\log \triangle)$-machine algorithm whereas EDF is not even an $o(\triangle)$-machine algorithm for this problem, so LLF is significantly better than EDF for the hard-real-time scheduling problem in this model.

We conclude by showing that no $(1+\epsilon)$-speed algorithm exists for the hard-real-time scheduling problem for $\epsilon < 1/4$, and that no $(1+\epsilon)$-speed algorithm exists for minimizing average flow time for $\epsilon < 1/10$. We prove these results by first giving simpler proofs for the weaker bounds $1/5$ and

1/21 respectively.

2.1 Faster machines

Theorem 2.1 *For $1 \leq \alpha \leq 2 - \frac{1}{m}$, SRPT is an α-speed $\frac{2-1/m}{\alpha}$-approximation algorithm for preemptively minimizing the sum of completion times on parallel machines.*

In particular, when $\alpha = 2 - 1/m$, we get the following result.

Corollary 2.2 *SRPT is a preemptive, $(2 - \frac{1}{m})$-speed algorithm for minimizing average flow times on parallel machines.*

When $\alpha = 1$, Theorem 2.1 implies that SRPT is a $(2 - \frac{1}{m})$-approximation algorithm which improves slightly a bound of 2 from [**?**].

Corollary 2.3 *SRPT is a $(2 - \frac{1}{m})$-approximation algorithm for sum of completion times on parallel machines.*

Theorem 2.4 *EDF is a preemptive, $(2 - \frac{1}{m})$-speed algorithm for hard-real-time scheduling on parallel machines.*

Theorem 2.5 *LLF is a preemptive, $(2 - \frac{1}{m})$-speed algorithm for hard-real-time scheduling on parallel machines.*

The key to the proof of all three theorems is the following fundamental relationship between machine speed and total work done by any busy algorithm such as SRPT, EDF, or LLF. 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 2.6 *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-α machines, $A(I, \beta t) \geq A'(I,t)$ for any algorithm A' using m speed-1 machines.*

Proof: We will prove this by contradiction. Fix an input instance I, and consider the set of "underworked" jobs: $\{J_i \mid \exists$ a time $t(i)$ such that $A(I, \beta t(i)) < A'(I, t(i))$ and A has done less work on job J_i by time $\beta t(i)$ than A' has done on job J_i by time $t(i)\}$. If the Lemma is false, this set must be nonempty. Let J_j be such a job from this set with a minimum release time r_j and let t denote $t(j)$. Thus, $A(I, \beta t) < A'(I,t)$ and, since A has done less work on job J_j by time βt than A' has done on J_j by time t,

$$A(j, \beta t) < A'(j,t). \tag{1}$$

Since J_j has a minimum release time r_j, it follows that $A(I, r_j) \geq A'(I, \frac{r_j}{\beta})$. Therefore, we can also conclude that the total work done by A from time r_j to time βt must be strictly less than the total work done by A' from time $\frac{r_j}{\beta}$ to time t. That is, we also have the following constraint.

$$A(I, \beta t) - A(I, r_j) < A'(I,t) - A'(I, \frac{r_j}{\beta}) \tag{2}$$

We will derive a contradiction by showing that constraints (1) and (2) cannot be true simultaneously.

We first analyze the algorithm A from time r_j to time βt. We divide this time period into 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 x denote the total length of the overloaded time intervals and y denote the total length of the underloaded time intervals. Obviously, $x + y = \beta t - r_j$.

With respect to x and y, we now derive a lower bound on the total amount of work done by A from time r_j to time βt and the amount of work done by A on job J_j by time βt. 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 $\alpha m x + \alpha y$ total work from time r_j to time βt. Furthermore, A does at least αy work on job J_j by time βt.

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. Furthermore, A' can do at most $t - r_j \leq t - \frac{r_j}{\beta}$ work on job J_j by time t.

Plugging these lower bounds for A and upper bounds for A' into constraints (1) and (2), we conclude that

$$m\left(t - \frac{r_j}{\beta}\right) > \alpha(mx + y) \tag{3}$$

and

$$t - \frac{r_j}{\beta} > \alpha y. \tag{4}$$

We now show that (3) and (4) cannot simultaneously be true. Add (3) to $(m-1)$ times (4) 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

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

This last inequality cannot hold since $\alpha\beta = 2 - \frac{1}{m}$ by definition. Thus, the result follows. □

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 to I using m speed-α machines. Then for any time t, the number of completed jobs in $SRPT(I)$ at time $t\frac{2-1/m}{\alpha}$ is at least as many as the number of completed jobs in $S(I)$ at time t. This implies that for all k, the completion time of the kth job completed in $SRPT(I)$ is no later than $\frac{2-1/m}{\alpha}$ times the completion time of the kth 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 2.6 and the definition of I_t, all $|I_t|$ jobs will be completed by time $\frac{2-1/m}{\alpha}t$ in $SRPT(I_t)$ given m speed-α machines. We now use the following lemma, proved in [**?**].

Lemma 2.7 *[**?**] If SRPT completes k jobs of input instance I' by any 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 $\frac{2-1/m}{\alpha}t$ in $SRPT(I)$. □

Proof of Theorem 2.4: 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 $\{J_1, \ldots, J_t\}$ for $1 \leq t \leq n$; that is, I_t consists of the t jobs in I with the t smallest deadlines (with some variation allowed due to tiebreaking). We also define I_0 to be the empty set of jobs. We prove by induction on t that EDF, using m speed-$(2-1/m)$ machines, legally schedules I_t for $0 \leq t \leq n$.

Clearly EDF legally schedules I_0 and I_1. Assume EDF legally schedules I_t for $0 \leq t \leq n-1$. 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} and note this set must contain at least job J_{t+1}. Then $I_{t+1} - Q$ must be I_q for some q where $0 \leq q \leq t$. By our induction hypothesis, EDF legally schedules I_q. Because EDF gives priority to jobs with earlier deadlines, $EDF(I_{t+1})$ is identical to $EDF(I_q)$ on the jobs in I_q. Thus, all jobs in I_q complete by their deadlines in $EDF(I_{t+1})$. By Lemma 2.6, 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. □

We now show this result is tight.

Lemma 2.8 *EDF, using m speed-$(2-\frac{1}{m}-\epsilon)$ machines, cannot schedule some input instances which can be scheduled on m speed-1 machines.*

Proof: Consider the following input instance consisting of $m+1$ jobs all released at time 0. The first m jobs have length $x\frac{m-1}{m}$ and deadlines x. The $m+1^{st}$ job has length x and deadline $x+1$. EDF will schedule the m smaller jobs first since they have earlier deadlines and will not complete them until time $\frac{x(m-1)}{m(2-\frac{1}{m}-\epsilon)}$. Thus, the completion time of the larger job will be $\frac{x(m-1)}{m(2-\frac{1}{m}-\epsilon)} + \frac{x}{2-\frac{1}{m}-\epsilon} = \frac{(2m-1)x}{2m-1-m\epsilon} = x + \frac{m\epsilon x}{2m-1-m\epsilon}$. So, if we choose $x > \frac{2m-1-m\epsilon}{m\epsilon}$, the completion time of the larger job will exceed $x+1$ and thus will fail to complete by its deadline. On the other hand, the optimal offline algorithm can schedule this input instance by devoting one machine to the larger job from time 0 to time x and $m-1$ machines to the remaining m jobs. □

We now will prove Theorem 2.5. We begin by defining some notation. Note, the notation is more general than is needed for this proof because this notation will also be used in section 2.2 when we consider LLF and extra machines.

Definition 2.1 *For any $m \geq 2$, an input instance I is is (c,s,m)*-hard *if LLF cannot legally schedule input instance I using cm speed-s machines while OPT can schedule I using m speed-1 machines.*

Definition 2.2 *We define the* failure time *in any (c,s,m)-hard input instance I using cm speed-s machines, $FT(I,c,s,m)$, to be the first time in LLF(I) where there are more than cm active jobs with laxity 0. We define this set of jobs to be the* failure set *of I, $FS(I,c,s,m)$.*

We wish to prove that there are no $(1, 2-1/m, m)$-hard input instances for any $m \geq 2$. Rather than working with an arbitrary $(1, 2-1/m, m)$-hard input instance, we will work with canonical input instances which consist only of "relevant jobs." That is, we will strip away any jobs that do not contribute to the failure of LLF to schedule the input instance.

Definition 2.3 *For any input instance I, and any two jobs J_i and J_j in I, we say that J_i* blocks *J_j if there exists a time t in LLF(I) where (i) both jobs are in the system and available for execution at time t and (ii) the laxity of job J_i is at most the laxity of job J_j at time t. Note a job J_i blocks itself.*

Definition 2.4 *For any job J_i in any input instance I, $B(J_i)$ denotes the set of jobs in I which block job J_i. For any set of jobs $Y \subset I$, $B(Y)$ denotes the set of jobs in I which block* any *job in Y.*

Definition 2.5 *For any input instance I and any set of jobs $Y \subseteq I$, define $R^1(Y) = B(Y)$. For $i \geq 2$, define $R^i(Y)$ as $B(R^{i-1}(Y))$. Finally define $R^*(Y) = \cup_{i \geq 0} B^i(Y)$.*

Fact 2.1 *For any input instance I, any subset $Y \subseteq I$, and any $c, s \geq 1$, the following statements hold for $R^*(Y)$.*

1. *$R^*(Y) \subseteq I$.*
2. *If I is feasible on cm speed-s machines, then $R^*(Y)$ is also feasible on cm speed-s machines.*
3. *$\triangle(I) \geq \triangle(R^*(Y))$.*
4. *Each job in $R^*(Y)$ is scheduled identically in LLF($R^*(Y)$) and LLF(I) assuming LLF uses cm speed-s machines in both cases.*

Definition 2.6 *If input instance $I = R^*(FS(I', c, s, m))$ for some (c, s, m)-hard input instance I', we say that I is a* canonical *(c, s, m)-hard input instance.*

Note, for any (c, s, m)-hard input instance I, there exists a corresponding *canonical* (c, s, m)-hard input instance CI which consists of $R^*(FS(I, c, s, m))$. Because of Fact 2.1, in particular property 3 of Fact 2.1, we can limit our attention to only canonical (c, s, m)-hard input instances.
Proof of Theorem 2.5: Suppose there exists a $(1, 2 - 1/m, m)$-hard input instance I'. Then there must exist a canonical $(1, 2 - 1/m, m)$-hard input instance I. We will show that there are no canonical $(1, 2 - 1/m, m)$-hard input instances I.

Let I denote any canonical $(1, 2 - 1/m, m)$-hard input instance. We first observe that OPT must do at least as much work on every job by the failure time $FT(I, 1, 2 - 1/m, m)$ as LLF does by $FT(I, 1, 2 - 1/m, m)$. This follows since every job J_j in I belongs to $R^*(FS(I, 1, 2 - 1/m, m))$ since I is a canonical instance. Therefore, every job in I either has a deadline before the failure time $FT(I, 1, 2 - 1/m, m)$ or has 0 laxity at $FT(I, 1, 2 - 1/m, m)$. This means that if any job receives less processing by time $FT(I, 1, 2 - 1/m, m)$ than it receives in $LLF(I)$, it either has missed its deadline or it has negative laxity. In either case, this job will not complete by its deadline.

We now observe that OPT must actually do more work on at least one job in $FS(I, 1, 2 - 1/m, m)$ by the failure time. This follows since there are at least $m + 1$ jobs with zero laxity at $FT(I, 1, 2 - 1/m, m)$, and OPT cannot possibly complete these jobs by their deadlines given only m speed-1 machines.

Putting these two observations together, they imply that OPT must do more total work than LLF by time $FT(I, 1, 2 - 1/m, m)$. However, by Lemma 2.6, LLF has done at least as much work as OPT has by time $FT(I, 1, 2 - 1/m, m)$. Thus, it follows that there cannot be a canonical $(1, 2 - 1/m, m)$-hard input instance, and thus there are no $(1, 2 - 1/m, m)$-hard input instances, and the result follows. □

Lemma 2.6 also has implications for on-line load balancing. In particular, Lemma 2.6 leads to the following result which is a generalization of Graham's proof that List Scheduling is a $(2 - \frac{1}{m})$-approximation algorithm for minimizing the makespan [8] of a schedule (makespan is the maximum completion time of any job in the schedule).

Corollary 2.9 *For $1 \leq \alpha \leq 2 - \frac{1}{m}$, List Scheduling is an α-speed $[(2 - \frac{1}{m})/\alpha]$-approximation algorithm for minimizing the makespan.*

2.2 Extra Machines

The key to our positive results for faster machines was Lemma 2.6. Unfortunately, the natural analogue to this result does not hold for extra machines as we shall show in a moment.

Definition 2.7 *Let A denote an arbitrary m-processor scheduling algorithm. For any $m \geq 2$, the family of input instances $I(m)$ is* c-lazy *for A if there exists a time $t \geq 0$ such that (i) A, given cm speed-1 machines, does not finish all jobs in $I(m)$ by time t, and (ii) all jobs in $I(m)$ can be finished by time t on m speed-1 machines. We refer to the first such time t as the* lazy point *or LP of $I(m)$.*

Note $I(m)$ is really a family of input instances parametrized by m, but we shall simply refer to $I(m)$ as a single input instance I. Also, when the algorithm A is clear from context, we simply use c-lazy rather than c-lazy for A.

Lemma 2.10 *Let A denote any busy algorithm which prioritizes shorter jobs over longer jobs. Then for any integer c, there exists an input instance I which is c-lazy for A.*

Proof: All jobs in I are released at time 0. Input I contains cm length-1 jobs and $\lfloor \frac{m}{2} \rfloor$ length-$2cm$ jobs. Since A gives priority to shorter jobs, the length-$2cm$ jobs do not complete until time $2cm+1$. However, the optimal m-machine schedule can finish all jobs by time $2cm$ by devoting $\lfloor \frac{m}{2} \rfloor$ machines to the length-$2cm$ jobs and the remaining machines to the length-1 jobs. □

We build upon Lemma 2.10 to derive strong lower bounds for both LLF and EDF for the hard-real-time scheduling problem.

Our first goal is to show that LLF is not a c-machine algorithm for any constant c for any $m \geq 2$. That is, we wish to show there exist $(c,1,m)$-hard input instances for any $m \geq 2$ and $c \geq 1$ as defined in Definition 2.1. For the remainder of this section, we will typically use c-hard input instance in place of $(c,1,m)$-hard input instance.

We build a c-hard input instance from the c-lazy input instance of Lemma 2.10 in two steps. The first step creates a c-lazy input instance $I(c,y,m)$ for LLF for any integer c by (i) setting deadlines and thus laxities to insure that LLF gives priority to shorter jobs and (ii) invoking more waves of short jobs to increase the amount of missing work at the lazy point (LP) of $I(c,y,m)$.

Definition 2.8 *Define the job set $I(c,y,m)$ as follows. First there are $\lfloor \frac{m}{2} \rfloor$ length-$2cy$ jobs with release times 0, deadlines $2cy+y$, and thus initial laxities of y. Meanwhile, for $0 \leq i \leq y-2c$, there are cm length-1 jobs with release times $2ci$, deadlines $2c(i+1)$, and thus laxities of $2c-1$. Altogether, $(y-2c+1)cm$ length-1 jobs are released from time 0 to time $2c(y-2c)$.*

Lemma 2.11 *For all $m \geq 2$, integers $c \geq 1$, and $y \geq 2c$, job set $I(c,y,m)$ is c-lazy for LLF. In particular, the $\lfloor \frac{m}{2} \rfloor$ length-$2cy$ jobs of $I(c,y,m)$ will each have $y-2c+1$ remaining processing time and $2c-1$ laxity at the lazy point $2cy$ in LLF$(I(c,y,m))$.*

Proof: See figure 2 for a pictoral proof. At time 0, $\lfloor \frac{m}{2} \rfloor$ jobs jobs are released, each of length $2cy$ and laxity y. By the assumption of this lemma $y \geq 2c$. There are also cm length-1 jobs released with laxity $2c-1$. Thus the length-1 jobs have smaller laxity and all will be run from time 0 to time 1. Thus all the long jobs will lose one unit of laxity before the release of the next set of length-1 jobs. In general, for $0 \leq i \leq y-2c$, at time $2ci$, the laxity of each of the length-$2cy$ jobs is exactly $y-i$ which is at least $2c$. The length-1 jobs released at these times have laxities $2c-1$. This implies that for $0 \leq i \leq y-2c$, at time $2ci$, LLF will choose to use all cm machines to schedule

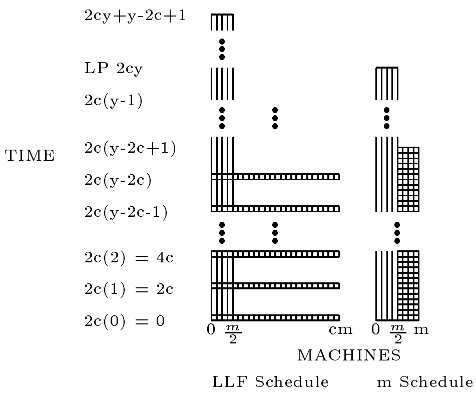

Figure 2: LLF and optimal m schedule for $I(c, y, m)$

the just released cm length-1 jobs. Thus the length-$2cy$ jobs will not be executed from time $2ci$ to time $2ci + 1$ for $0 \leq i \leq y - 2c$, a total of $y - 2c + 1$ time units. Therefore, at time $2cy$, the $\lfloor \frac{m}{2} \rfloor$ length-$2cy$ jobs will still each require $y - 2c + 1$ units of processing and will have laxities $2c - 1$.

On the other hand, all jobs can be completed by time $2cy$ on m machines if the following schedule is used. Dedicate $\lfloor \frac{m}{2} \rfloor$ machines to the length-$2cy$ jobs from time 0 to time $2cy$. Use the other machines to schedule the length-1 jobs. $\square$

We now convert the c-lazy input instance $I(c, y, m)$ into a c-hard input instance $I(c, m)$. The key idea is that LLF still has significant work to do in the time interval $[2cy, 2cy + y]$ whereas the optimal m-machine algorithm has no work to do after the lazy point $2cy$. Thus, we overload LLF by repeating scaled down versions of this c-lazy input instance *within* the time interval $[2cy, 2cy + y]$. To simplify the definition of the final c-hard input instance $I(c, m)$, we first augment our definition of the c-lazy input instance $I(c, y, m)$ to include a global release time t.

Definition 2.9 *Define the job set $I(t, c, y, m)$ as follows. There are $\lfloor \frac{m}{2} \rfloor$ length-$2cy$ jobs with release times t, deadlines $t + 2cy + y$, and thus initial laxities of y. Meanwhile, for $0 \leq i \leq y - 2c$, there are cm length-1 jobs with release times $t + 2ci$, deadlines $t + 2c(i + 1)$, and thus laxities of $2c - 1$. Altogether, $(y - 2c + 1)cm$ length-1 jobs are released from time t to time $t + 2c(y - 1)$.*

Definition 2.10 *See figure 3 for a graphical representation of this input instance. We define the input instance $I(c, m)$ to be $\cup_{i=1}^{6c} I(t_{i-1}, c, y_i, m)$ where t_i and y_i are defined as follows. Let $y_1 = 4(2c - 1)(2c + 1)^{6c-1}$ (the key property is that $y_1 > 2(2c - 1)(2c + 1)^{6c-1}$). We recursively define $y_i = y_{i-1}/(2c + 1)$ for $2 \leq i \leq 6c$ (note $y_{i-1} = (2c + 1)y_i$ and $y_1 = (2c + 1)^{6c-1}y_{6c}$). Let $t_i = 2c\sum_{j=1}^{i} y_j$ for $0 \leq i \leq 6c$ (note $t_0 = 0$).*

Lemma 2.12 *Consider input $I(c, m)$ scheduled by LLF and by OPT. At time t_i for $1 \leq i \leq 6c$, LLF will still have $i\lfloor \frac{m}{2} \rfloor$ 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 released before time t_i by time t_i.*

Proof: We prove the result by induction on i. The base case where $i = 1$ follows directly from Lemma 2.11.

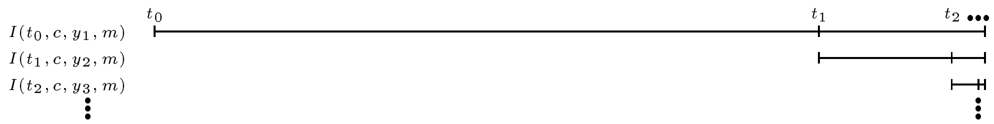

Figure 3: Graphical representation of $I(c, m)$

The inductive step is proven as follows. By the inductive hypothesis, LLF, run with cm machines, has $i\lfloor\frac{m}{2}\rfloor$ 'leftover" jobs with at least $y_i - 2c + 1$ required units of processing and laxities at most $2c - 1$ at time t_i for some $1 \leq i \leq 6c - 1$ whereas the optimal m machine schedule has completed all jobs released before time t_i by time t_i.

If we assume these $i\lfloor\frac{m}{2}\rfloor$ leftover jobs receive continuous processing from time t_i to time t_{i+1} (that is, these jobs receive $2cy_{i+1}$ units of processing), these $i\lfloor\frac{m}{2}\rfloor$ jobs will have $y_i - 2cy_{i+1} - 2c + 1 = y_{i+1} - 2c + 1$ (note $y_i = (2c+1)y_{i+1}$) required units of processing remaining and laxities $2c - 1$ at time t_{i+1}.

If we assume these leftover jobs do not "steal" any processing time from any jobs released at or after time t_i, Lemma 2.11 tells us the $\lfloor\frac{m}{2}\rfloor$ length-$2cy_{i+1}$ jobs released at time t_i will have $y_{i+1} - 2c + 1$ required units of processing remaining and laxities $2c - 1$ at time t_{i+1}. Combining these jobs with the $i\lfloor\frac{m}{2}\rfloor$ leftover jobs gives us $(i+1)\lfloor\frac{m}{2}\rfloor$ jobs which have at least $y_{i+1} - 2c + 1$ required units of processing remaining and laxities $2c - 1$ at time t_{i+1}.

Meanwhile, Lemma 2.11 tells us that the optimal m machine schedule completes all the jobs released in the time interval $[t_i, t_{i+1})$ by time t_{i+1}. Thus, the Lemma is proven. □

Theorem 2.13 *LLF is not a c-machine algorithm for hard-real-time scheduling for any constant c.*

Proof: From Lemma 2.12, we see that input instance $I(c, m)$ can be feasibly scheduled on m speed-1 machines. Meanwhile, at time $t_{6c} = 2c\sum_{j=1}^{6c} y_j$, LLF will still have at least $2cm$ jobs with at least $y_{6c} - 2c + 1$ required units of processing remaining and laxities of at most $2c - 1$. If $y_{6c} > 2(2c-1)$, LLF will not be able to legally schedule this instance. Working backwards, we see that since we defined $y_1 = 4(2c-1)(2c+1)^{6c-1}$, $y_{6c} = 4(2c-1) > 2(2c-1)$, and the theorem follows. □

Note that $\triangle(I(c, m))$, the ratio of longest job length to shortest job length, is large. We now show that any c-hard input instance I for LLF must have a structure similar to that of $I(c, m)$; in particular $\triangle(I)$ must be exponential in c. More specifically, we will show that any c-hard input instance I for LLF consists of at least c c-lazy building blocks I_i for $1 \leq i \leq c$ for LLF where block I_i arrives after the lazy point of the previous building block. That is, input I must look something like Figure 3. We will use this to show the exponential nature of $\triangle(I)$.

In order to prove the desired result, we will work with canonical $(c, 1, m)$-hard input instances as defined in Definition 2.6. For the remainder of this section, we will refer to such input instances as canonical c-hard input instances. Because of Fact 2.1, in particular property 3 of Fact 2.1, we can limit our attention to only canonical c-hard input instances.

Our first result about any canonical c-hard input instance CI is that at the failure time $FT(CI, c, s, m)$, the optimal m-machine schedule has done at least as much work on every job in CI as LLF has using cm machines. Note, this result does not hold for an arbitrary c-hard input instance I since I may contain "extraneous" jobs on which LLF has done more work by

$FT(I,c,s,m)$ than the optimal m-machine schedule. For the remainder of the section, we will use $FT(I,c)$ to denote $FT(I,c,s,m)$ and $FS(I,c)$ to denote $FS(I,c,s,m)$.

Definition 2.11 *Let $A(j,t)$ denote the amount of processing specified by algorithm A for job J_j by time t.*

Deficit $def(j,t)$ denotes how much extra processing OPT, using m machines, has done on job J_j by time t than LLF has done on J_j using cm machines.

Definition 2.12 *Let OPT be the optimal algorithm for scheduling on m speed-1 machines. Then for any canonical c-hard input instance CI, $def(j,t) = OPT(j,t) - LLF(j,t)$. Furthermore, for any subset $Y \subseteq CI$, $def(Y,t) = \sum_{J_j \in Y} def(j,t)$.*

Lemma 2.14 *Let CI denote any canonical c-hard input instance. OPT must do at least as much work on all jobs in CI as LLF does by the failure time of CI; that is, $\forall J_j \in CI$ $def(j, FT(CI,c)) \geq 0$.*

Proof: Since CI is a canonical c-hard input instance, every job J_j in CI belongs to $R^*(FS(CI,c))$. Therefore, every job J_j in CI either has a deadline before the failure time $FT(CI,c)$ or has 0 laxity at $FT(CI,c)$. This means that if any job J_j receives less processing by time $FT(CI,c)$ than it receives in $LLF(CI)$, it either has missed its deadline or it has negative laxity. In either case, this job will not complete by its deadline. $\square$

We now use Lemma 2.14 to prove that CI must be similar to $I(c,m)$. We first identify the arrival times and lazy points of at least c c-lazy building blocks. We do this by identifying the latest time $t_i \leq FT(CI,c)$ when there are at most im available jobs. Time interval $[t_{i-1}, t_i)$ bounds the arrival time and lazy point of at least one c-lazy building block for $1 \leq i \leq c$. Each building block contains at most m jobs in $FS(CI,c)$, and these jobs in $FS(CI,c)$ are the incomplete jobs at times t_i.

Definition 2.13 *For any canonical c-hard input instance CI, define time t_i for $0 \leq i \leq c$ to be the minimum of $\{t \mid$ There are at least $im+1$ available jobs in $LLF(CI)$ from time t to time $FT(CI,c).\}$.*

Definition 2.14 *For any canonical c-hard input instance CI and for $1 \leq i \leq c$, define T_i to be the time interval $[t_{i-1}, t_i)$ and T'_i to be the time interval $[t_i, FT(CI,c)]$.*

See figure 4 for a graphical depiction of these concepts.

Definition 2.15 *For any canonical c-hard input instance CI and for any time interval T, let $J(T)$ denote the set of jobs in CI which arrive during time interval T.*

The following three statements, while not necessarily true, provide intuition for the remainder of our proof:

1. The jobs in $J(T_i)$ for $1 \leq i \leq c$ not completed by time t_i are in $FS(CI,c)$.
2. The work deficit of these jobs is no more than $|T_i|$ because T_i contains the time interval when these jobs accumulate their work deficits.
3. This work deficit, and thus $|T_i|$, is an upper bound on $|T'_i|$.

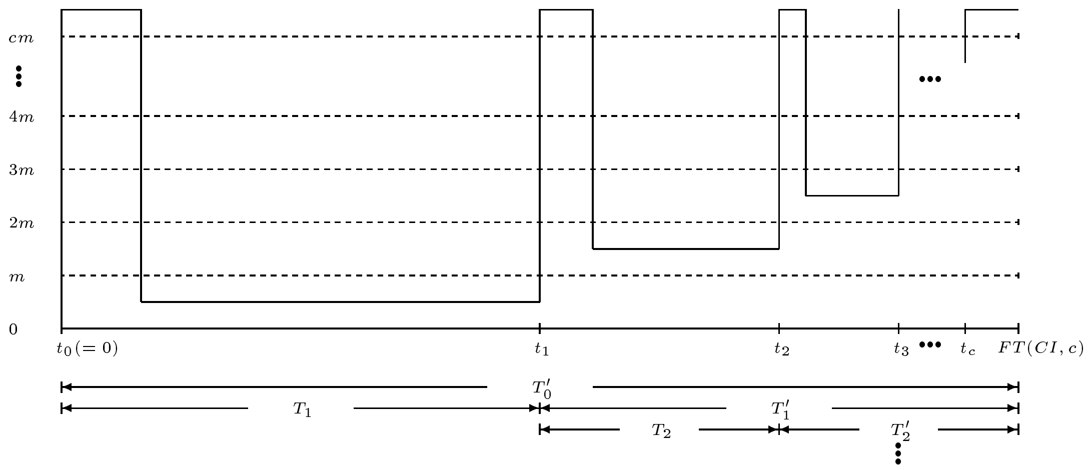

Figure 4: Number of available jobs in $LLF(CI)$ over time.

Unfortunately, we are not able to directly relate the work deficit of jobs in $J(T_i)$ to $|T'_i|$. Instead we prove the following result relating these work deficits to $|T'_{i+1}|$.

Lemma 2.15 *For any canonical c-hard input instance CI,*

$$def(J(T_i), t_i) > m|T'_{i+1}| \quad \textit{for } 1 \le i \le c-1. \tag{5}$$

It then follows that $|T_i| > |T'_{i+1}| \quad$ for $1 \le i \le c-1$.

Proof: We first observe that $def(J(T_i), t_i) \le m|T_i| \quad$ for $1 \le i \le c$ since the jobs in $J(T_i)$ do not arrive before time t_{i-1} and OPT can perform at most $m|T_i|$ units of processing on them by time t_i. Clearly, combining this inequality with inequality (5) yields $|T_i| > |T'_{i+1}|$. Thus, all that remains is proving inequality (5).

From Lemma 2.14, we first observe that $def(J(T'_{i-1}), FT(CI,c)) \ge 0$. We now show this implies (5). For $1 \le i \le c-1$, throughout time interval T'_{i-1}, at most $(i-1)m$ of the available jobs arrived before time t_{i-1}. This implies that during time interval $T_{i+1} = [t_i, t_{i+1})$, at least $m+1$ of the machines are always working on jobs that belong to $J(T'_{i-1})$. Likewise, during time interval $T'_{i+1} = [t_{i+1}, FT(CI,c)]$, at least $2m+1$ machines are doing work on jobs that belong to $J(T'_{i-1})$. Since OPT has only m machines total, these observations imply $def(J(T'_{i-1}), t_{i+1}) \le def(J(T'_{i-1}), t_i) - |T_{i+1}|$ and $def(J(T'_{i-1}), FT(CI,c)) \le def(J(T'_{i-1}), t_{i+1}) - (m+1)|T'_{i+1}|$. Combining these two inequalities with $def(J(T'_{i-1}), FT(CI,c)) \ge 0$, we conclude that $def(J(T'_{i-1}), t_i) \ge |T_{i+1}| + (m+1)|T'_{i+1}|$. We now observe that $def(J(T'_{i-1}), t_i) = def(J(T_i), t_i)$ since $J(T_i)$ contains all jobs in $J(T'_{i-1})$ released before time t_i. Thus we have $def(J(T_i), t_i) \ge |T_{i+1}| + (m+1)|T'_{i+1}|$ which implies inequality (5), and the Lemma follows. $\square$

We now prove our main upper bound result for LLF.

Theorem 2.16 *Consider any canonical c-hard input instance CI. Then $\triangle(CI) = \Omega(\phi^{c-\lceil\sqrt{2c}\rceil})$ where ϕ is the golden ratio.*

Proof: We first use Lemma 2.15 to show that $L_i = |T'_{c-i}|$ grows faster than the Fibonacci numbers. For $2 \le i \le c$,

$$L_i \;=\; |T'_{c-i}|$$

$$
\begin{aligned}
&= FT(CI,c) - t_{c-i} \\
&= (FT(CI,c) - t_{c-(i-1)}) + (t_{c-(i-1)} - t_{c-i}) \\
&= L_{i-1} + |T_{c-(i-1)}| \\
&> L_{i-1} + |T'_{c-(i-2)}| \\
&= L_{i-1} + L_{i-2}.
\end{aligned}
$$

The key step of this derivation is our use of Lemma 2.15 in replacing $|T_{c-(i-1)}|$ with $|T'_{c-(i-2)}|$.

We next show that (i) $\triangle(CI) > |T'_2| = L_{c-2}$ and that (ii) $|T'_{c-\lceil\sqrt{2c}\rceil}| = L_{\lceil\sqrt{2c}\rceil} \geq 1$. Clearly, if we prove both of these bounds, it follows that $\triangle(CI) > F_{(c-2)-(\lceil\sqrt{2c}\rceil-1)}$ where F_i is the i^{th} Fibonacci number ($F_0 = 0$ and $F_1 = 1$). Because $F_n = \left\lfloor \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor \approx \frac{\phi^n}{\sqrt{5}}$ where ϕ is the golden ratio ≈ 1.61803, transitivity implies that $\triangle(CI) > \frac{\phi^{c-\lceil\sqrt{2c}\rceil-1}}{\sqrt{5}} - \frac{1}{2}$ which means $\triangle(CI) = \Omega(\phi^{c-\lceil\sqrt{2c}\rceil})$ and the result follows.

We first prove the lower bound on $\triangle(CI)$. Let $S \subseteq J(T_1)$ denote the at most m available jobs in $LLF(CI)$ just prior to time t_1. Since LLF has completed all other jobs in $J(T_1)$, it follows that $def(S,t_1) \geq def(J(T_1),t_1)$. On the other hand, since no single job can have a work deficit greater than its length, $m\triangle(CI) \geq def(S,t_1)$ as S contains at most m jobs. Applying transitivity, $m\triangle(CI) \geq def(J(T_1),t_1)$. We then use (5) to observe that $def(J(T_1),t_1) > m|T'_2| = mL_{c-2}$. Finally, applying transitivity and dividing by m, we get the desired result that $\triangle(CI) > L_{c-2}$.

We now prove the lower bound on $L_{\sqrt{2c}}$. We will show that some job must arrive no earlier than time $t_{c-\lceil\sqrt{2c}\rceil}$ and be completed by LLF no later than time t_c. Thus its length must be $\leq t_c - t_{c-\lceil\sqrt{2c}\rceil} \leq L_{\lceil\sqrt{2c}\rceil}$, and the lower bound follows.

We first observe that just prior to time t_c, there are at most cm jobs in the system. We now show that more than cm jobs arrive during the time interval $[t_c - t_{c-\lceil\sqrt{2c}\rceil})$ which yields the lower bound. Consider each subinterval T_i for $c - \lceil\sqrt{2c}\rceil + 1 \leq i \leq c$ within $[t_c - t_{c-\lceil\sqrt{2c}\rceil})$. Inequality (5) shows that LLF falls behind OPT in each subinterval T_i with respect to the jobs released during that subinterval, and this can only happen if the system is overloaded. Therefore, during each of these subintervals, there must be a time when there are at least $cm+1$ available jobs. Note that at most $(i-1)m$ jobs were in the system at time t_{i-1}. Therefore, at least $(c-i+1)m+1$ jobs must have arrived during subinterval T_i. That is, $|J(T_i)| \geq (c-i+1)m+1$. Summing up over all subintervals, we observe that $\sum_{i=c-\lceil\sqrt{2c}\rceil+1}^{c} |J(T_i)| > cm$. Therefore, more than cm jobs were released during the time interval $[t_c - t_{c-\lceil\sqrt{2c}\rceil})$ and the lower bound on $L_{\sqrt{2c}}$ follows. □

Corollary 2.17 *LLF is an $O(\log \triangle)$-machine algorithm for hard-real-time scheduling.*

Proof: Follows from Theorem 2.16. □

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

Theorem 2.18 *EDF is not a $(\left\lfloor \triangle \frac{m-1}{m} \right\rfloor)$-machine algorithm for hard-real-time scheduling.*

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

The input instance $I(m,\triangle)$ consists of cm jobs with deadlines of $2\triangle$ and execution times of 2 and one job with a deadline of $2\triangle + 1$ and an execution time of $2\triangle$ (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 $I(m, \triangle)$ 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 $I(m, \triangle)$, however, is clearly schedulable on m machines as we can put the one job with deadline $2\triangle + 1$ 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. $\square$

2.3 General lower bounds

In this section, we 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 $\epsilon < 1/21$. We also show that no $(1+\epsilon)$-machine algorithm exists for the hard-real-time scheduling problem for $\epsilon < 1/4$ and for minimizing average flow time for $\epsilon < 1/10$. The key to these proofs is constructing instances where any on-line algorithm given insufficient extra resources completes less work by certain critical deadlines than the optimal offline algorithm given m speed-1 machines.

We first consider the hard-real-time scheduling problem, and we use the following two input instances. Input instance IN contains m length-1 jobs with laxity 1 at the time of their release and $\frac{m}{2}$ length-2 jobs with laxity 2 at the time of their release. The release times of these jobs is 0. Input instance IN' is IN plus m length-1 jobs released at time 1 with zero laxity at the time of their release.

Lemma 2.19 *Let A denote any $(1+\epsilon)$-speed algorithm where $\epsilon < 1/5$ or any $(1+\epsilon)$-machine algorithm where $\epsilon < 1/4$. Then A does not finish the length-2 jobs in IN by time 2.*

Proof: We first observe that IN' can be legally scheduled on m speed-1 machines which implies that A must legally schedule IN'. Simply devote all m machines to the m length-1 jobs of IN in time interval $[0, 1]$, then devote all m machines to the newly arrived length-1 jobs in time interval $[1, 2]$, and finally devote $\frac{m}{2}$ machines to the $\frac{m}{2}$ length-2 jobs in time interval $[2, 4]$.

We now bound the amount of processing that A can give the length-2 jobs of IN' in the time interval $[0, 1]$. Since IN and IN' are indistinguishable until time 1, this bound also applies to IN. The m extra jobs of IN' will use m of the $(1+\epsilon)m$ available units of work in the time interval $[1, 2]$ leaving only ϵm units of work that can be devoted to the length-1 jobs of IN during the time interval $[1, 2]$. Thus, during time interval $[0, 1]$, A must complete $(1-\epsilon)m$ units of processing on these jobs. Since A can perform at most $(1+\epsilon)m$ units of work during any unit time interval, this implies that A can devote at most $2\epsilon m$ units of work to the $\frac{m}{2}$ length-2 jobs in the time interval $[0, 1]$.

We now bound the amount of processing that the length-2 jobs of IN can receive in the time interval $[1, 2]$. Here we need to treat faster machines separately from extra machines, and thus we achieve distinct lower bounds.

Given m faster machines, the maximum amount of processing each length-2 job can receive in the time interval $[1, 2]$ is $(1+\epsilon)$ which means that altogether these length-2 jobs receive $\frac{m}{2}(1+\epsilon)$ units of process in time interval $[1, 2]$. Combining this with the $2\epsilon m$ units of work they might receive in time interval $[0, 1]$, these length-2 jobs receive at most $\frac{m}{2} + \frac{5\epsilon}{2}m$ units of processing in the time interval $[0, 2]$. If $\epsilon < \frac{1}{5}$, the total remaining processing time for these jobs at time 2 is $m - (\frac{m}{2} + \frac{5\epsilon}{2}m)$ which is greater than 0, and the faster machine result follows.

The analysis for extra machines is similar. The difference is that in the time interval $[1, 2]$, the maximum amount of processing each length-2 job can receive is 1 which means that altogether these length-2 jobs receive $\frac{m}{2}$ units of processing in time interval $[1, 2]$. Thus, the total amount of processing received by the length-2 jobs in time interval $[0, 2]$ is $\frac{m}{2} + 2\epsilon m$. If $\epsilon < \frac{1}{4}$, the total remaining processing time for these jobs at time 2 is $m - (\frac{m}{2} + 2\epsilon m)$ which is greater than 0, and and the extra machine result follows. $\square$

We digress for a moment to explain in more detail the main difference between faster machines and extra machines. Faster machines can "catch up" after they have made a mistake whereas extra machines cannot. In the above example, if the input is IN rather than IN', it is a mistake to perform $m - \epsilon$ units of work on the length-1 jobs in time interval $[0, 1]$ because then in time interval $[1, 2]$, the best any algorithm can do is devote $\frac{m}{2}$ machines to the length-2 jobs while devoting the remaining machines to the remaining ϵm units of processing required by the length-1 jobs. Thus many of these other machines will be idle for much of this time interval. In the extra-machine case, no extra work can be done on the length-2 jobs in the time interval $[1, 2]$, but in the faster-machine case, some extra work, ϵ units to be exact, can be done on each length-2 job in time interval $[1, 2]$.

We now describe how we exploit the fact that no faster-machine or extra-machine algorithm finishes all jobs in IN by time 2 given insufficient extra resources. Currently, the length-2 jobs receive at most $2\epsilon m$ units of processing in time interval $[0, 1]$, and they have δm unfinished work at time 2. Suppose we release a second copy of IN at time 2. We will show that the length-2 jobs released at time 2 receive at most $(2\epsilon - \delta)m$ units of processing in time interval $[2, 3]$. Furthermore, the amount of unfinished work these length-2 jobs have at time 4 will be at least $2\delta m$. If we continue to release new copies of IN, we can show that this ripple effect continues until the length-2 jobs released at time $2t$ for some $t \geq 0$ receive no processing in time interval $[2t, 2t + 1]$. The following lemma formalizes this intuitive argument.

First, we need to name these input instances consisting of repeated copies of IN. Let I_i for $i \geq 1$ denote the input instance consisting of i copies of IN where copy j is released at time $2(j - 1)$. Let I'_i for $i \geq 1$ denote the input instance I_i plus m length-1 jobs with 0 laxity released at time $2i - 1$, one time unit after the last jobs are released in I_i.

Lemma 2.20 *Let A denote any $(1 + \epsilon)$-speed algorithm where $\epsilon < 1/5$ or any $(1 + \epsilon)$-machine algorithm where $\epsilon < 1/4$. Consider input instance I_n where $n \geq 1$. For any $i \leq n - 1$, let δm denote the amount of unfinished work the length-2 jobs released at time $2(i - 1)$ have at time $2i$. Then A does at most $\max(0, (2\epsilon - \delta)m)$ work on the length-2 jobs released at time $2i$ in time interval $[2i, 2i + 1]$.*

Proof: We first observe that I'_{i+1} can be legally scheduled on m speed-1 machines, so A must legally schedule I'_{i+1}. We now bound the amount of processing that the length-2 jobs released at time $2i$ of I'_{i+1} can receive in the time interval $[2i, 2i + 1]$. Since I'_{i+1} and I_n are indistinguishable until time $2i + 1$, this bound also applies to I_n. The analysis is almost identical to that of Lemma 2.19.

First, the jobs available at time $2i$ contain $(1 + \delta)m$ units of work which must be completed by time $2i + 2$ with m units coming from the length-1 jobs released at time $2i$ and δm units coming from the length-2 jobs released at time $2(i - 1)$. The m zero-laxity jobs of I'_{i+1} released at time $2i + 1$ will use m of the $(1 + \epsilon)m$ available units of work in the time interval $[2i + 1, 2i + 2]$ leaving only ϵm units of work that can be devoted to the remaining $(1 + \delta)m$ units of work with deadline $2i + 2$. Thus, during time interval $[2i, 2i + 1]$, A must complete $(1 + \delta - \epsilon)m$ units of processing on these jobs. Since A can perform at most $(1 + \epsilon)m$ units of work during any unit time interval, this implies that A can devote at most $\max(0, (2\epsilon - \delta)m)$ units of work to the $\frac{m}{2}$ length-2 jobs released at time $2i$ in the time interval $[2i, 2i + 1]$. $\square$

We are now ready for the main hard-real-time scheduling result.

Theorem 2.21 *There is no on-line $(1+\epsilon)$-speed algorithm for hard-real-time scheduling for $m \geq 2$ and even and $\epsilon < 1/5$, and there is no on-line $(1+\epsilon)$-machine algorithm for hard-real-time scheduling for $m \geq 2$ and even and $\epsilon < 1/4$.*

Proof: Let A denote any $(1+\epsilon)$-speed algorithm where $\epsilon < 1/5$ or any $(1+\epsilon)$-machine algorithm where $\epsilon < 1/4$. From Lemmas 2.19 and 2.20, it is clear that there exists some input I_n with $n \geq 0$ such that A does no work on the length-2 jobs released at time $2(n-1)$ in time interval $[2(n-1), 2n-1]$. This follows because the minimum amount of unfinished work in the length-2 jobs two units after their arrival grows by a fixed amount with each new copy of IN until this quantity exceeds $2\epsilon m$.

We now prove the faster-machine result. We augment I_n by releasing m length-2 jobs with zero laxity at time $2n$. Since A did not schedule the length-2 jobs released at time $2(n-1)$ during time interval $[2(n-1), 2n-1]$, these jobs still require at least $\frac{1-\epsilon}{2}m$ units of work during time interval $[2n, 2n+2]$. Adding in the jobs released at time $2n$, a total of $2m + \frac{1-\epsilon}{2}m$ units of work need to be completed during time interval $[2n, 2n+2]$ which exceeds the $2(1+\epsilon)m$ units of processing available when $\epsilon < 1/5$. Thus A does not legally schedule this augmented input instance, but this augmented input instance can clearly be legally scheduled on m speed-1 machines. Thus A is not a $(1+\epsilon)$-speed algorithm for $\epsilon < 1/5$, and the faster machine result follows.

We now prove the extra machine result. The analysis is nearly identical. We again augment I_n by releasing m length-2 jobs with zero laxity at time $2n$. Since A did not schedule the length-2 jobs released at time $2(n-1)$ during time interval $[2(n-1), 2n-1]$, these jobs still require at least $\frac{m}{2}$ units of work during time interval $[2n, 2n+2]$. Adding in the jobs released at time $2n$, a total of $\frac{5}{2}m$ units of work need to be completed during time interval $[2n, 2n+2]$ which exceeds the $2(1+\epsilon)m$ units of processing available when $\epsilon < 1/4$. Thus, A does not legally schedule this augmented input instance, so A is not a $(1+\epsilon)$-machine algorithm, and the extra machine result follows. □

We now prove lower bounds for minimizing average flow time.

Theorem 2.22 *There is no on-line $(1+\epsilon)$-speed algorithm for scheduling to minimize flow times for $\epsilon < 1/21$ when the number of machines m is an even number ≥ 2.*

Proof: The proof is based on the following adversary strategy. Define $\hat{1} = \frac{1}{1+\epsilon}$ and $\hat{2} = \frac{2}{1+\epsilon}$. At time 0, m length-1 jobs and $m/2$ length-2 jobs are released. Between time 0 and time $\hat{1}$, an on-line algorithm using m speed-$(1+\epsilon)$ machines can do at most m units of work. Let mx denote the amount of work the on-line algorithm completes on the length-2 jobs before time $\hat{1}$, and note that $0 \leq x \leq 1/2$ since there are only $m/2$ length-2 jobs. If $x \leq 5/21$, then m length-1 jobs are released at time $\hat{2}$ (scenario A). Otherwise, m length-1 jobs are released at time $\hat{1}$ (scenario B).

Let us first analyze scenario A. In analyzing any on-line algorithm, we will focus on three distinct time intervals: $[0, \hat{1})$, $[\hat{1}, \hat{2})$, and $[\hat{2}, \infty)$. During $[0, \hat{1})$, no jobs complete, so the flow time incurred by any on-line algorithm is $\frac{3m}{2}\hat{1}$. During $[\hat{1}, \hat{2})$, no length-2 jobs complete, so the flow time incurred by any on-line algorithm is at least the total remaining execution time of the length-1 jobs $(mx\hat{1})$, plus the total incremental flow time of the length-2 jobs $(\frac{m}{2}\hat{1})$.

Since no new jobs arrive after time $\hat{2}$, SRPT is now the optimal strategy. Each length-2 job receives at least 1 unit of processing ($\hat{1}$ units of time) during $[0, \hat{2})$ because each length-1 job can run at most $\hat{1}$ time. Thus the remaining length-2 jobs have remaining length at most 1 and have priority no lower than the newly-arrived length-1 jobs in the SRPT schedule. If all the length-2 jobs ran during all of $[\hat{1}, \hat{2}]$, then there would be a total of $m/2$ work done during that time interval. Therefore, the length-2 jobs have at least $\frac{m}{2} - mx$ total remaining units of work at time $\hat{2}$.

Since the length-2 jobs are scheduled without delay in some optimal schedule, the total incremental flow time of the length-2 jobs in $[\hat{2},\infty)$ is their total execution time which is at least $(\frac{m}{2} - mx)\hat{1}$. Each length-2 job delays exactly one newly arrived length-1 job, so the total delay incurred by the newly arrived length-1 jobs in $[\hat{2},\infty)$ is the total execution time of the length-2 jobs which is $(\frac{m}{2} - mx)\hat{1}$. Adding this to their total execution time of $m\hat{1}$, the total flow time incurred by the newly arrived length-1 jobs in $[\hat{2},\infty)$ is at least $(\frac{m}{2} - mx)\hat{1} + m\hat{1}$.

Adding all flow time terms for all intervals together yields a total flow time of at least $(4-x)m\hat{1}$. Plugging in the maximum value of x which is $\frac{5}{21}$, this reduces to $\frac{79}{21}m\hat{1}$.

Meanwhile, the optimal speed-1 offline algorithm will initially devote $\frac{m}{2}$ machines to the length-2 jobs from time 0 to time 2 and devote the other $\frac{m}{2}$ machines to the length-1 jobs from time 0 to time 2, thus incurring a flow time of $\frac{5m}{2}$ from time 0 to time 2 for these jobs. Finally, it will devote m machines to the jobs released at time $\hat{2}$ from time 2 to time 3 incurring a flow time of $m(1+2-\hat{2})$ $= m + \frac{2\epsilon m}{(1+\epsilon)}$. Thus, the flow time incurred by the optimal algorithm is $\frac{7m}{2} + \frac{2\epsilon m}{1+\epsilon}$. Comparing this to on-line's flow time of at least $\frac{79}{21}m\hat{1}$, we see that on-line's flow time exceeds offline's flow time if $\epsilon < \frac{1}{21}$.

We now consider scenario B. In this case, when we analyze any on-line algorithm, we will focus only on two time intervals: $[0,\hat{1})$ and $[\hat{1},\infty)$. During $[0,\hat{1})$, no jobs complete, so the flow time incurred by any on-line algorithm is $\frac{3m}{2}\hat{1}$.

Since no new jobs will arrive after time $\hat{1}$, SRPT is now the optimal strategy which means that the remaining original length-1 jobs have highest priority, the newly arrived length-1 jobs have second highest priority, and the original length-2 jobs have lowest priority (they each could have received at most 1 unit of processing by time $\hat{1}$).

Since each original length-1 job is scheduled without delay, their total incurred flow time in $[\hat{1},\infty)$ is their total execution time which is $mx\hat{1}$. The execution of each original length-1 job delays the start of one of the newly arrived length-1 jobs. Thus the total delay incurred by the newly arrived length-1 jobs during $[\hat{1},\infty)$ is at least $mx\hat{1}$. Adding this to their total execution time of $m\hat{1}$, the total flow time incurred by the length-1 jobs during $[\hat{1},\infty)$ is at least $mx\hat{1} + m\hat{1}$. Finally, no original length-2 jobs can start before time $\hat{2}$ since they cannot start until the first newly arrived length-1 job finishes, and this cannot happen before $\hat{2}$. Thus, the total delay incurred by the length-2 jobs in $[\hat{1},\infty]$ is at least $\frac{m}{2}(\hat{2} - \hat{1}) = \frac{m}{2}\hat{1}$. Adding this to their total remaining execution time of $(m - mx)\hat{1}$, the total flow time incurred by the length-2 jobs during $[\hat{1},\infty]$ is at least $\frac{m}{2}\hat{1} + (m - mx)\hat{1}$.

Summing together all terms from interval $[\hat{1},\infty)$ and the flow time for interval $[0,\hat{1})$ yields a total flow time of at least $(4+x)\hat{1}m$. Plugging in the minimum value of x which is $\frac{5}{21}$, this reduces to $\frac{89}{21}m\hat{1}$.

Meanwhile, the optimal offline algorithm will devote m machines from time 0 to time 1 to the length-1 jobs, m machines from time 1 to time 2 for the newly arrived length-1 jobs, and $\frac{m}{2}$ machines from time 2 to time 4 for the length-2 jobs. Thus, the flow time incurred by the offline algorithm will be $m + m(1+1-\hat{1}) + \frac{m}{2}4 = 4m + \frac{\epsilon m}{1+\epsilon}$. Comparing this to on-line's flow time of at least $\frac{89m}{21}\hat{1}$, we see that on-line's flow time exceeds offline's flow time if $\epsilon < \frac{1}{21}$.

Thus, no matter what on-line does before time $\hat{1}$, it cannot guarantee an optimal flow time if $\epsilon < \frac{1}{21}$. Thus, the theorem follows. $\square$

Theorem 2.23 *There is no on-line $(1+\epsilon)$-machine algorithm for scheduling to minimize flow times for $\epsilon < 1/10$ when the number of machines m is an even number ≥ 2.*

Proof: The proof is very similar to that of Theorem 2.22. The adversary strategy is slightly modified as follows. At time 0, m length-1 jobs and $m/2$ length-2 jobs are released. Between time 0 and time 1, an on-line algorithm using $(1+\epsilon)m$ speed-1 machines can do at most $(1+\epsilon)m$ units of work. Let xm be the amount of work the on-line algorithm devotes to the length-2 jobs before time 1 and note that $0 \leq x \leq 1/2$. If $x \leq 3/10$, then m length-1 jobs are released at time 2 (scenario A). Otherwise, m length-1 jobs are released at time 1 (scenario B).

In scenario A, we focus on three distinct time intervals: $[0,1)$, $[1,2)$, and $[2,\infty)$. Our analysis is similar to the analysis of scenario A in Theorem 2.22. During time $[0,1)$, no jobs complete, which contributes $3m/2$ to the flow time. If xm work is devoted to the length-2 jobs in $[0,1)$, then $(1+\epsilon-x)m$ work can be devoted to the length-1 jobs. Thus, during $[1,2)$ the incremental flow time is $m/2$ for the incomplete length-2 jobs and the $(x-\epsilon)m$ remaining work from the length-1 jobs. As before, SRPT is an optimal strategy starting at time 2, so the remaining length-2 jobs have priority and have total remaining length at least $m/2 - mx$. However, now the longest ϵm remaining length-2 jobs will not delay any newly-released jobs (the new jobs will run on the extra machines). These longest jobs are each at most length 1 and therefore the newly-arrived jobs are delayed by a total of at least $m/2 - xm - \epsilon m$ time. Thus the incremental flow time in $[2,\infty)$ is $m/2 - mx$ for the remaining processing on the length-2 jobs, $m/2 - mx - \epsilon m$ for the forced idle time of the new jobs, and m for the processing of the new jobs. Thus the total flow time incurred by any on-line algorithm for scenario A is at least $(4 - x - 2\epsilon)m$. Plugging in the maximum value of x which is $\frac{3}{10}$, this reduces to $(\frac{37}{10} - 2\epsilon)m$. Meanwhile, the flow time incurred by the optimal algorithm is $\frac{7m}{2}$. Comparing this to on-line's flow time of at least $(\frac{37}{10} - 2\epsilon)m$, we see that on-line's flow time exceeds offline's flow time if $\epsilon < \frac{1}{10}$.

We now consider scenario B. In this case, when we analyze any on-line algorithm, we will focus only on two distinct time intervals: $[0,1)$ and $[1,\infty)$. During $[0,1)$, no jobs complete until at least time 1, so the flow time incurred by any on-line algorithm in $[0,1)$ is $\frac{3m}{2}$. At time 1, the shortest jobs are the remaining original length-1 jobs followed by the new length-1 jobs followed by the remnants of the original length-2 jobs (they each could have received at most 1 unit of processing by time 1). Since there are $\frac{3m}{2}$ original length-2 jobs and new length-1 jobs and only $(1+\epsilon)m$ machines, this implies that each of the original length-1 jobs will delay at least one of the original length-2 jobs or one of the new length-1 jobs. Furthermore, $\frac{m}{2} - m\epsilon$ of the original length-2 jobs will be delayed by the new length-1 jobs for a full time unit. Thus, the flow time incurred by employing the SRPT strategy at time 1 is at least $m(x-\epsilon) + m(x-\epsilon) + m + (\frac{m}{2} - m\epsilon) + (m - mx)$ where the first term is the incremental flow time of the original length-1 jobs, the second term is the delay these jobs cause for other jobs, the third term is the processing time of the new length-1 jobs, the fourth term is the delay incurred by the original length-2 jobs as they wait for the new length-1 jobs to finish, and the fifth term is the remaining processing time of the length-2 jobs. Thus, the total flow time incurred by any on-line algorithm for scenario B is at least $(4 + x - 3\epsilon)m$. Plugging in the minimum value of x which is $\frac{3}{10}$, this reduces to $(\frac{43}{10} - 3\epsilon)m$.

Meanwhile, the optimal offline algorithm will devote m machines from time 0 to time 1 to the length-1 jobs, m machines from time 1 to time 2 for the new length-1 jobs, and $\frac{m}{2}$ machines for the length-2 jobs from time 2 to time 4. Thus, the flow time incurred by the offline algorithm will be $4m$. Comparing this to on-line's flow time of at least $(\frac{43}{10} - 3\epsilon)m$, we see that on-line's flow time exceeds offline's flow time if $\epsilon < \frac{1}{10}$.

Thus, no matter what on-line does before time 1, it cannot guarantee an optimal flow time if $\epsilon < \frac{1}{10}$. Thus, the theorem follows. $\square$

3 On-line preemptive weighted flow time

In this section we give an on-line 2-speed algorithm for scheduling preemptively to minimize average *weighted* flow time on a *single* processor. This algorithm builds upon the work of Hall et. al. [10] which uses a variety of linear programming formulations to design approximation algorithms to minimize $\sum w_j C_j$. 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 ρ-approximation algorithm for $\sum w_j C_j$ in no way is guaranteed to be a ρ-approximation algorithm for $\sum w_j F_j$. No non-trivial polynomial-time algorithm is known for this problem. We first present an offline algorithm and then convert it into an on-line algorithm.

Our algorithm is a simple greedy algorithm; at any time t, it schedules the available job with highest priority. We now define how our algorithm assigns priorities to jobs and what an available job is.

We use the *mean busy time* of a job for a given schedule, defined, for example, in [6]. Given a schedule for which job j ends at time T, let $I_j(t) = 1$ if job j is running at time t and let $I_j(t) = 0$ otherwise. Then the mean busy time M_j is the average of all the times when job j is processing:

$$M_j = \frac{1}{p_j} \int_{r_j}^{T} I_j(t)dt.$$

To determine priorities, we compute the optimal solution to the following linear program R given job set I and denote this solution $\overline{M} = (\overline{M}_1, \ldots, \overline{M}_n)$. Job i has priority over job j if $\overline{M}_i < \overline{M}_j$.

R:

Minimize $\sum_{j=1}^{n} w_j(M_j + \frac{1}{2}p_j)$ subject to

$$\sum_{j \in S} p_j M_j \geq p(S)(r_{\min}(S) + \frac{1}{2}p(S)) \quad \text{for each } S \subseteq I \tag{6}$$

$$\tag{7}$$

where

$$\begin{aligned} p(S) &= \sum_{j \in S} p_j \\ r_{\min}(S) &= \min_{j \in S} r_j. \end{aligned}$$

The following lemma, proved in [**?**], shows that the computed solution $\overline{M} + \frac{1}{2}p_j$ is a lower bound on the optimal solution.

Lemma 3.1 *[**?**] Given a preemptive schedule for instance I on one machine, let $C_1, \ldots, C_n$ denote the job completion times in this schedule, and let $M_1, \ldots, M_n$ denote the mean busy times in this schedule. Then the M_j for $1 \leq j \leq n$ satisfy the inequalities given by (6) and $C_j \geq M_j + \frac{1}{2}p_j$.*

While R has an exponential number of constraints, Goemans et. el show [**?**] show the following greedy algorithm computes an optimal solution to R: starting at time zero, always schedule the available job j with highest w_j/p_j. Job j is available if it has been released and has not been completed. This schedule can be computed in $O(n \log n)$ time for n jobs, by sorting the jobs by release date, and maintaining a priority queue keyed in w_j/p_j. We must update the priority queue

O(n) times (each time a job is released or completed). Given this schedule, we can compute the mean busy time of each job in O(n) total time, since there are only O(n) total job pieces.

Our algorithm defines an available job at time t to be a job J_j such that $r_j \leq t$ *and* the amount of processing received by job J_j at time t is no more than $p_j/2$; that is, our algorithm only processes *one-half* of each job. Clearly, if our algorithm is given a speed-2 machine, it will have exactly the same behavior and fully process each job. We call this algorithm Preemptively-Schedule-Halves-by-$\overline{M}_j$.

Lemma 3.2 *Let $\overline{M}_1 \leq \cdots \leq \overline{M}_n$ be an optimal solution to R, and let $\widetilde{C}_1, \ldots, \widetilde{C}_n$ denote the completion times of the job halves found by* Preemptively-Schedule-Halves-by-$\overline{M}_j$. *Then, for $1 \leq j \leq n$, $\widetilde{C}_j \leq \overline{M}_j$.*

Proof: Let $I_j = \{J_1, \ldots, J_j\}$. Let t be the latest time before $\widetilde{C}_j$ at which the machine is not processing a job in I_j just before time t. If no such time exists, $t = 0$. Let $S_j \subseteq I_j$ denote the set of jobs that are at least partially processed in the interval $[t, \widetilde{C}_j]$.

We first show that $\widetilde{C}_j \leq r_{\min}(S_j) + \frac{1}{2}p(S_j)$ for $1 \leq j \leq n$. We observe that all jobs in S_j were released at time t or later; otherwise, they would be running just before time t since the jobs in I_j and thus S_j have priority over all other jobs. Futhermore, at least one of the jobs $J_l \in S_j$ begins processing at time t, and the processor was not working on any job in I_j just prior to time t. Since the jobs in S_j have priority over all jobs not in I_j, this implies $r_j = t$ which implies $t = r_{\min}(S_j)$. Since the processor is always processing a job in S_j from time t to time $\widetilde{C}_j$ and it only schedules half of each job, we conclude $\widetilde{C}_j \leq r_{\min}(S_j) + \frac{1}{2}p(S_j)$.

We now will prove that $\overline{M}_j \geq r_{\min}(S_j) + \frac{1}{2}p(S_j)$ for $1 \leq j \leq n$. Because $S_j \subseteq I_j$, for all $k \in S_j$, $\overline{M}_k \leq \overline{M}_j$. Thus, $\overline{M}_j p(S_j) \geq \sum_{k \in S_j} p_k \overline{M}_k$. From inequality (6) applied to set S_j, we get $\sum_{k \in S_j} p_k \overline{M}_k \geq r_{\min}(S_j)p(S_j) + \frac{1}{2}p(S_j)^2$. Transitivity gives us $\overline{M}_j p(S_j) \geq r_{\min}(S_j)p(S_j) + \frac{1}{2}p(S_j)^2$, which leads to $\overline{M}_j \geq r_{\min}(S_j) + \frac{1}{2}p(S_j)$. Thus $\widetilde{C}_j \leq \overline{M}_j$, as we wished to show. □

We convert Preemptively-Schedule-Halves-by-$\overline{M}_j$ 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 offline algorithm because the relative priorities of jobs that arrive before any time t are unaffected by the arrival of jobs after time t [7].

Theorem 3.3 Preemptively-Schedule-Halves-by-$\overline{M}_j$ *is an on-line 2-speed algorithm for scheduling preemptively to minimize $\sum w_j F_j$ on 1 machine.*

Proof: This follows from Lemmas 3.1 and 3.2. □

4 On-line nonpreemptive scheduling

In this section, we consider the significantly-more-difficult problem of nonpreemptive scheduling. We focus on the problems of minimizing total flow time and minimizing total *weighted* flow time, though we do consider the problem of nonpreemptive scheduling with deadlines as well. We give an on-line $O(\log \Delta)$-machine algorithm for minimizing total flow time, an $O(\log \Delta)$-machine 2-approximation algorithm for minimizing total weighted flow time, and an 8-speed $O(\log \Delta)$-machine algorithm for the hard-real-time scheduling problem. We then give an input instance in which the gap between the optimal c-machine nonpreemptive schedule's flow time (for any constant c) and the optimal 1-machine preemptive schedule's flow time is polynomial in the number of jobs in the input instance. This example shows that the analysis of any cm-machine nonpreemptive algorithm for constant c will require a stronger lower bound on the optimal nonpreemptive flow time on m machines.

4.1 Algorithms

We construct extra-machine nonpreemptive algorithms for minimizing flow time and weighted flow time. We first give a 2-machine algorithm for minimizing flow time if $\triangle(I) \leq 2$, and we give a 2-machine 2-approximation algorithm for minimizing weighted flow time if $\triangle(I) \leq 2$. We then use these special algorithms to construct $O(\log \Delta)$-extra-machine algorithms which are optimal for arbitrary input instances.

4.1.1 Input instances with similar sized jobs

In this section, we will only consider input instances where $\triangle(I) \leq 2$. Furthermore, we assume that $p_{\max}$ and $p_{\min}$, the longest and shortest jobs in I, are known a priori.

The following algorithms will be used in the design or analysis of our on-line algorithms:

- O - the optimal nonpreemptive m-machine algorithm,
- $\tilde{O}$ - the optimal preemptive m-machine algorithm,
- G - the greedy algorithm which, when a machine is idle, schedules any available job, and
- Gp - a modified greedy algorithm that, when a machine becomes idle, schedules the available job with the largest weight at the next time that is an integer multiple of p.

The schedules produced by these algorithms on an input instance I will be denoted $O(I), \tilde{O}(I), G(I)$, and $Gp(I)$, respectively. We use the notation $F_j^{X(I)}$ to denote the flow time of job J_j when algorithm X is run on input I, and $S_j^{X(I)}$ to denote the starting time of job J_j when algorithm X is run on input I.

We first consider the unweighted case where $w_j = 1$ for $1 \leq j \leq n$.

Lemma 4.1 *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 $\sum_j F_j^{G(I)} = \sum_j F_j^{\tilde{O}(I)}$.*

Proof: The greedy nonpreemptive algorithm G is optimal even if preemption is allowed, 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. □

We now consider the case when $\triangle(I) \leq 2$.

Lemma 4.2 *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_j. There is an on-line algorithm U, run on $2m$ machines, that given input I produces a schedule with $\sum_j F_j^{U(I)} \leq \sum_j F_j^{\tilde{O}(I)}$.*

Proof: Let I' denote a modified instance of I where all $p_j = p$. Note, p is known a priori. Now consider an optimal preemptive schedule for instance I. If we remove the last $p_j - p$ units of each job J_j from the schedule, the flow time of job J_j will be reduced by at least $p_j - p$ and we will obtain a feasible, though not necessarily optimal, schedule for instance I'. Therefore, we have that

$$F^{\tilde{O}(I')} \leq F^{\tilde{O}(I)} - \sum_j (p_j - p). \tag{8}$$

By Lemma 4.1, $G(I')$ has no preemptions, but it is an optimal preemptive schedule for I'.

We now show that given $2m$ machines, we can create an on-line schedule $U(I)$ such that $S_j^{U(I)} = S_j^{G(I')}$ for $1 \leq j \leq n$. We do this by using two machines to process the jobs assigned to one machine in $G(I')$; more specifically, jobs which are assigned to a specific machine in $G(I')$ are alternately placed on the two corresponding machines in $U(I)$. Because the processing times at most double when changing from I' to I, the alternate placement insures that no jobs overlap.

Since $S_j^{U(I)} = S_j^{\tilde{O}(I')}$ for $1 \leq j \leq n$, we have

$$\begin{aligned} F^{U(I)} &= \sum_j S_j^{U(I)} + \sum_j p_j - \sum_j r_j \\ &= \sum (S_j^{\tilde{O}(I')} + p - r_j) + \sum_j (p_j - p) \\ &= F^{\tilde{O}(I')} + \sum_j (p_j - p) \\ &\leq F^{\tilde{O}(I)}, \end{aligned}$$

where the last inequality follows from Equation 8.

□

We now consider minimizing weighted flow time. 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 show that G_p is a 2-approximation.

Lemma 4.3 *Let p be a positive integer. Let I be a flow time problem with $p_j = p$ for all jobs j. Then*

$$\sum_j w_j F_j^{Gp(I)} \leq \sum_j w_j (F_j^{O(I)} + p) \leq 2 \sum_j w_j F_j^{O(I)}$$

.

Proof: Let $Op(I)$ be an optimal schedule in which each job is required to start at a time that is an integral multiple of p. We first observe that $\sum_j w_j F_j^{Op(I)} \leq \sum_j w_j (F_j^{O(I)} + p)$. To see this, transform $O(I)$ by moving each job later so that it begins at an integral multiple of p. Clearly this transformed schedule is valid, each job begins at an integral multiple of p, and the completion time of each job has increased by no more than p.

We now show that $\sum_j w_j F_j^{Gp(I)} = \sum_j w_j F_j^{Op(I)}$. Observe that as long as we are restricting ourselves to schedules in which each job starts at a multiple of p, we may round all release dates up to the next multiple of p without changing the set of feasible schedules. We thus have an input instance in which all $p_j = p$ and all r_j are multiples of p, and hence we can divide all p_j and r_j by p to obtain an instance for which $p_j = 1$ and r_j are integral. This scales the objective function by exactly p also. For this problem it is known that the greedy algorithm is optimal, so $\sum_j w_j F_j^{Gp(I)} = \sum_j w_j F_j^{Op(I)} \leq \sum_j w_j (F_j^{O(I)} + p)$.

Since $F_j^{\mathrm{O(I)}} \geq p$ for all j, this means $\sum_j w_j (F_j^{O(I)} + p) \leq 2 \sum_j w_j F_j^{O(I)}$. Thus the result follows.
□

We now use a proof similar to that of Lemma 4.2 to bound the performance of an on-line $(2m)$-machine version of Gp, on the input instances where $p \le p_j \le 2p$ for some p.

Lemma 4.4 *Let p be a positive integer. Let I be an input instance with $p \le p_j \le 2p$. There is an on-line algorithm U that, given $2m$ machines, produces a schedule $U(I)$ such that $\sum_j w_j F_j^{U(I)} \le 2\sum_j w_j F_j^{O(I)}$.*

Proof: Form I' from I by rounding the processing times down to p. Note, p is known a priori. Using the two machines with alternating job placement technique of Lemma 4.2, we create an on-line schedule $U(I)$ where $S_j^{U(I)} = S_j^{Gp(I')}$ for $1 \le j \le n$. Since all jobs in I' have processing time exactly p, this implies that $S_j^{Gp(I')} = F_j^{Gp(I')} - p + r_j$ for $1 \le j \le n$. Furthermore, from Lemma 4.3, $\sum_j w_j F_j^{Gp(I')} \le \sum_j w_j (F_j^{O(I')} + p)$.

Combining these bounds, we get that

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

□

Note all the results of this subsection also hold with speed-2 machines instead of doubling the number of machines. However, the extra-machine results of the next subsection cannot be transformed into equivalent faster-machine results.

4.1.2 Input instances with arbitrarily-sized jobs

We now give algorithms for input instances I where $\triangle(I)$ is arbitrarily large. We again first consider unweighted flow time and then consider weighted flow time. We split the jobs into groups of similarly-sized jobs and put each group on its own machine. The number of groups is logarithmic in $\Delta(I)$, 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 $\triangle$ is defined as $\max p_j / \min p_j$, the ratio between the minimum and maximum processing times.

Theorem 4.5 *There is an on-line $2\lceil \log \triangle \rceil$-machine algorithm for minimizing total flow time and an on-line $2\lceil \log \triangle \rceil$-machine 2-approximation algorithm for minimizing total weighted flow time* *assuming $p_{\min}$ and $p_{\max}$ are known a priori. If $p_{\min}$ and $p_{\max}$ are not known, the extra machine constant changes from $2\lceil\log\triangle\rceil$ to $2(\lceil\log\triangle\rceil+1)$ for both total flow time and total weighted flow time.*

Proof: We divide the jobs into $\lceil\log\triangle\rceil$ groups, where the ith group contains all the jobs with $2^{i-1}p_{\min} \le p_j \le 2^i p_{\min}$ where each group is assigned $2m$ machines. Within each group, processing times differ by at most a factor of 2, so Lemmas 4.2 and 4.4 can be applied. Since we assume $p_{\min}$ and $p_{\max}$ are known a priori, the first two results follow. For the unweighted case, this means our nonpreemptive algorithm uses $2m\lceil\log\triangle\rceil$ machines to achieve the the optimal *preemptive m-machine* flow-time.

If $p_{\min}$ and $p_{\max}$ are not known a priori, we use the processing time p_1 of the first job released to establish our groups. That is, the division points will be at $2^i p_1$ where i may be negative since p_1 may not be the smallest job. Given that p_1 may not be $2^i p_{\min}$ for some integral value of i, we potentially need one extra group of $2m$ machines to insure we span $p_{\min}$ and $p_{\max}$, and the final result follows. □

If we do not have $2(\lceil\log\triangle\rceil+1)m$ machines, we can adapt the algorithm for minimizing flow time into an $O(\log n)$-machine $(1+o(1))$-approximation algorithm.

Corollary 4.6 *There is an on-line $O(\log n)$-machine $(1+o(1))$-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: First assume we know $p_{\max}$ and n a priori and that we have $(6\log n)m+1$ machines. We now utilize only the $3\log n$ largest groups. More precisely, we devote $2m$ machines for jobs in the ranges

$$(p_{\max}, p_{\max}/2), (p_{\max}/2, p_{\max}/4), \ldots, (2p_{\max}/n^3, p_{\max}/n^3)$$

for a total of $(6\log n)m$ machines. We schedule these $3\log n$ groups optimally using Lemma 4.2. The final machine is devoted to the remaining small jobs. The total amount of processing required by these small jobs 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.

We now remove the assumption that $p_{\max}$ and n are known a priori. At any point in time, we use the largest processing time seen so far as our estimate of $p_{\max}$. We also use p_1, the processing time of the first job, to define the division points for our groups as in Theorem 4.5. This increases the number of groups needed from $3\log n$ to $3\log n+1$. Note we do not need an estimate of n. We simply cover as many groups as we have machines. If $p_{\max}$ is revised upwards at any time and certain job groups are reassigned to the small job category, then the $2m$ machines which had been assigned to those jobs finish processing any jobs currently being run and then are reassigned.

Suppose we have at least $(6\log n+2)m+1$ machines (even though we do not know the actual value of n). Then each time the estimate of $p_{\max}$ is revised, each large job waits at most $p_{\max}/n^3$ time for the machine to finish processing the small job it is working on. Clearly, the estimate of $p_{\max}$ can change at most n times. Thus, the total extra flow time for the large jobs incurred by these changes in the estimate of $p_{\max}$ is at most $n^2 p_{\max}/n^3 = p_{\max}/n$. Again, this multiplies the total flow time by a $1+o(1)$ factor and the first result follows.

To derive the second result, suppose we use speed-$(\frac{n}{n-2})$ machines. This means the largest job now only takes $p_{\max}\frac{n-2}{n} = p_{\max} - \frac{2p_{\max}}{n}$ time to process. Since the jobs are scheduled nonpreemptively, the flow time of this largest job is reduced by $\frac{2p_{\max}}{n}$ which is enough to absorb the total flow time of the small jobs and the delays the small jobs impose on the large jobs. □

Unfortunately, this technique does not work for weighted flow time 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.

Lemma 4.7 *Nonpreemptive EDF is an 8-speed algorithm for the hard-real-time scheduling problem when $p_{max} \leq 2p_{\min}$.*

Proof: We first note that Theorem 2.4 tells us that that preemptive EDF is a 2-speed algorithm for the hard-real-time scheduling problem in general, so it is also a 2-speed algorithm for the special case where $p_{\max} \leq 2p_{\min}$. To simplify notation, we use p to represent $p_{\min}$. We now show that this implies that nonpreemptive EDF is a 8-speed algorithm for the hard-real-time scheduling problem where $p_{\max} \leq 2p$. Note, we shall assume that preemptive EDF and nonpreemptive EDF have consistent ways for breaking ties when two or more jobs have identical deadlines.

For any input instance I, let $E(I)$ denote the schedule produced by preemptive EDF using m speed-2 machines, let $M_j^{E(I)}$ denote the time when the second half of job j begins being processed in $E(I)$, let $N(I)$ denote the schedule produced by nonpreemptive EDF using m speed-8 machines, and let $S_j^{N(I)}$ denote the start time of job j in $N(I)$. Our goal is to show that for all jobs j, $S_j^{N(I)} \leq M_j^{E(I)}$. This proves our result as the completion time of job j in $N(I)$ is exactly $S_j^{N(I)}+p_j/8$ whereas the completion time of job j in $E(I)$ is at least $M_j^{E(I)} + (p_j/2)/2 \geq S_j^{N(I)} + p_j/4$.

Renumber jobs in nondecreasing order of $M_j^{E(I)}$ breaking ties by deadlines with earlier deadlines having higher priority. Consider any job j and assume that $S_i^{N(I)} \leq M_i^{E(I)}$ for all $i < j$. We will show that $S_j^{N(I)} \leq M_j^{E(I)}$.

Since $M_j^{E(I)} \geq r_j + p_j/4$, if $S_j^{N(I)} \leq r_j + p_j/4$, we have $S_j^{N(I)} \leq M_j^{E(I)}$. Now consider the time interval $[r_j + p/4, S_j^{N(I)})$ in schedule $N(I)$. During this time period, all m machines must be busy or else job j would be scheduled earlier. Furthermore, all jobs being processed during this time interval must have higher priority than job j. This follows as the only reason a job j' with lower priority than job j would be running and job j would not is that j' was already running at time r_j. Since jobs have length at most $2p$ and nonpreemptive EDF has speed-8 machines, any job running at time r_j must be finished by time $r_j + p/4$.

Consider any job j' processed in $N(I)$ from time $r_j+p/4$ to time $S_j^{N(I)}$. Since $S_{j'}^{N(I)} \leq M_{j'}^{E(I)}$, job j' cannot be completed before time $S_{j'}^{N(I)}+p_{j'}/8$ in $E(I)$ as one half of job j' is unprocessed at time $S_{j'}^{N(I)}$ and we assume preemptive EDF only has speed-2 machines. Thus, job j' would have priority over job j in $E(I)$ in time interval $[S_{j'}^{N(I)}, S_{j'}^{N(I)} + p_j/8]$. Since this holds for all jobs j' processed in $N(I)$ in time interval $[r_j + p/4, S_j^{N(I)})$ and all processors are occupied in this time interval in $N(I)$, this implies that job j receives no processing in $E(I)$ in time interval $[r_j + p/4, S_j^{N(I)})$.

Therefore, preemptive EDF could only schedule job j in time interval $[r_j, r_j + p/4)$ prior to time $S_j^{N(I)}$. Since preemptive EDF has only speed-2 machines, this means that at least half of job j remains to be processed at time $S_j^{N(I)}$. This means that $S_j^{N(I)} \leq M_j^{E(I)}$ and the result follows.
□

Theorem 4.8 *There is a nonpreemptive 8-speed $\lceil \log \triangle \rceil$-machine algorithm for the hard-real-time scheduling problem when $p_{\min}$ and $p_{\max}$ are known a priori. If $p_{\min}$ and $p_{\max}$ are not known a priori, the extra machine constant changes from $\lceil \log \triangle \rceil$ to $\lceil \log \triangle \rceil + 1$.*

Proof: Let I be a feasible input instance given m speed-1 machines. Divide the jobs in I into $\lceil \log \triangle \rceil$ groups where the ith group contains all jobs with $2^{i-1} p_{\min} \leq p_j \leq 2^i p_{\min}$ and each group is assigned m speed-8 machines. Within each group, processing times differ by at most a factor of 2, so Lemma 4.7 applies which means we can schedule these jobs so that all complete by their deadlines using nonpreemptive EDF. Since we assume $p_{\min}$ and $p_{\max}$ are known a priori, the first result follows. If $p_{\min}$ and $p_{\max}$ are not known a priori, we use the processing time p_1 of the first job released to establish our groups. We potentially need one extra group of m speed-8 machines to insure we span $p_{\min}$ and $p_{\max}$, and the second result follows. $\square$

4.2 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 [?], 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 4.9 *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 show that the optimal preemptive flow time for $I(c, N)$ on one machine is at most $2(c + 1)N$ assuming $c/n \leq 1/2$; note the sum of the lengths of all jobs is $(c + 1)N$. To simplify the proof, we analyze the shortest processing time (SPT) algorithm rather than the optimal shortest remaining processing time (SRPT) algorithm. We will show that SPT finishes all jobs j by time $r_j + 2p_j$. The upper bound of $2(c + 1)N$ on the flow time for SPT will then follow. The SPT algorithm is simpler to analyze because a type l job always has priority over a type i job for $0 \leq i < l \leq c$.

Consider any job j released at time t, and assume that any job j' with $r_{j'} < r_j$ and $p_{j'} \leq p_j$ finishes by time $r_{j'} + 2p_{j'}$. We will show that job j completes by time $r_j + 2p_j$. Let job j be a type i job where $0 \leq i \leq c$. We first observe that one type l job is also released at time t for $i < l \leq c$. This follows from the structure of the input instance. We next observe that the previously released type l job for $i < l \leq c$ was released at time $t - n\ len(l)$. By our assumption, this job completes by time $t - (n-2)len(l)$. Since $c/n \leq 1/2$ and $c \geq 1$, this means $n \geq 2$ which means that this previously released type l job completes by time t. Therefore the only jobs that can delay job j during time interval $[t, t+2p_j]$ are the higher-priority jobs released during this time interval.

We now show that the total processing time of all jobs released during time interval $[t, t+2p_j)$ with higher priority than job j is exactly p_j. Combining this with our previous observation that all higher priority jobs released before time t complete by time t, we conclude that job j will complete by time $t + 2p_j$.

We again observe that one type l job is also released at time t. Furthermore, a second type l job is released at time $t + 2p_j$ for $i < l \leq c$. These two facts along with the fact that type l jobs are released every $n\ len(l)$ time units imply that the total processing time of all type l jobs released in the time interval $[t, t+2p_j) = \frac{2p_j}{n}$ for $i < l \leq c$. Thus, during time interval $[t, t+2p_j)$, the total processing time of higher priority jobs is exactly $2(c-i)p_j/n$ for $0 \leq i \leq c$. This is no more than $2cp_j/n$ which is no more than p_j since $c/n \leq 1/2$. Thus the upper bound of $2(c+1)N$ on SPT's 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 $k+1$.

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 $\Theta(Nn)$, since these jobs are released evenly through the interval, and therefore at least half of them will wait $len(k-1)/2$ units. Therefore, one of these jobs must complete before the end of this time interval. Since these jobs arrived after the beginning of the interval, this implies there exists a time interval from time 0 to time Nn during which $k+1$ jobs, one type i job for $0 \leq i \leq k$, is running simultaneously for $len(k) = len((k+1)-1)$ time. Thus, we have shown the property holds for $k+1$ as well and the inductive case is proven.

Therefore, we know either the optimal nonpreemptive schedule has flow time at least Nn or there must be some $len(c-1)$-time interval prior to time Nn when c jobs execute simultaneously on the c machines. Therefore, the $\frac{Nn}{len(c-1)}$ 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 $\Omega(Nn)$. $\square$

We now show that having faster machines as well as extra machines does not fundamentally change the result. Thus the logarithmic-machine speed-$(1+o(1))$ algorithm of Corollary 4.6 produces a nonpreemptive schedule with flowtime potentially polynomially better than the optimal speed-2 schedule for any constant number of machines.

Corollary 4.10 *There exists a family of input instances $I(c_1, c_2, N)$ with $\Theta(N)$ jobs such that the optimal nonpreemptive flow time for input instance $I(c_1, c_2, N)$ on c_1 speed-c_2 machines is $\Omega(N^{\frac{1}{2^{c_1+1}-2}}/c_2^2)$ times greater than the optimal preemptive flow time for input instance $I(c_1, c_2, N)$ on one speed-1 machine, for large enough N.*

Proof: We define $I(c_1, c_2, N) = I(c_1, N)$ and the proof of Theorem 4.9 applies mostly unchanged. The only difference the speed-c_2 machines makes is that we now focus on intervals of length $len(i)/c_2$ rather than intervals of length $len(i)$ when analyzing the nonpreemptive algorithm. The key property of instance $I(c_1, c_2, N)$ is now that during any time interval of length $len(i)/c_2$ between time 0 and time Nn, $\frac{Nn}{len(i)c_2}$ type-$(i+1)$ jobs arrive. This introduces a c_2^2 factor in the denominator of the optimal nonpreemptive flow time since we now have c_2 fewer jobs and the time they are in the system is reduced by a factor of c_2 as well. $\square$

We close this section on nonpreemptive scheduling by observing that the nonpreemptive setting is fundamentally different than the preemptive setting with respect to extra resource results. In particular, while additional speed is always as good as additional machines in the preemptive setting, this is not the case in nonpreemptive scheduling.

Theorem 4.11 *There exists an input instance I such that the optimal nonpreemptive flow time given one speed-c machine where $c \leq n^{1/4-\epsilon}$ and $\epsilon > 0$ is polynomially larger than the optimal preemptive flow time given one speed-1 machine.*

Proof: We utilize the input instance from [**?**]. In this input, there is a single job of length n which is released at time 0 and n jobs of length 1 where the ith such job is released at time $i\sqrt{n}$ for $1 \leq i \leq n$.

For this set of jobs, the optimal preemptive schedule is to run the large job whenever it is the only job in the system and to preempt it to run a short job whenever they arrive. It is easy to see this results in a flow time of $2n + \sqrt{n} + 1$ as the large job will be delayed by $\sqrt{n} + 1$ of the short jobs.

Now consider a nonpreemptive algorithm given one speed-c machine. If the big job is not run to completion before time $n\sqrt{n}$, it alone incurs a flow time of at least $n\sqrt{n}$ which is $\Theta(\sqrt{n})$ times the optimal preemptive flow time. On the other hand, if it runs to completion before time $n\sqrt{n}$, it requires n/c time to run to completion during which $\frac{n}{c\sqrt{n}}$ jobs are released. The total flow time incurred by these jobs as they wait for this long job to complete will be $\Theta\left(\frac{n}{c\sqrt{n}}\frac{n}{c}\right) = \Theta\left(\frac{n^{3/2}}{c^2}\right)$. Thus, if $c = n^{1/4-\epsilon}$ for $\epsilon > 0$, then the total flow time incurred by these delayed size-1 jobs will be $\Theta(n^{1+\epsilon^2})$ which is $\Theta(n^{\epsilon^2})$ times the optimal preemptive flow time. $\square$

Now consider the optimal nonpreemptive schedule for this input instance given two speed-1 machines. One machine runs the big job and the other machine runs all the length-1 jobs. The flow time is actually $2n$ which is better than the optimal preemptive flow time on one machine. Thus, this example demonstrates that in the nonpreemptive setting, unlike the preemptive setting, faster machines can be much less effective than extra machines.

5 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 5.1 *If A is an s-speed ρ-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 (ρs)-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 $1 \leq i \leq n$. 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. 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. □

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

Corollary 5.2 *For the problem of minimizing average flow time on a single processor when execution times are unknown until they complete, the Balance algorithm [12] is an s-stretch $\left(s + \frac{s}{s-1}\right)$-approximation algorithm when $1 \leq s \leq 2$ [12], and it is an s-stretch 2-approximation algorithm when $s \geq 2$ [?].*

Corollary 5.3 *SRPT is a $(2 - 1/m)$-stretch $(2 - 1/m)$-approximation algorithm for minimizing average flow time on multiple processors.*

Corollary 5.4 *The algorithm* Preemptively-Schedule-Halves-by-$\overline{M}_j$ *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 4 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] Michael Bender, Soumen Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms*, pages 270–279, 1998.

[2] S. Chakrabarti, C. Phillips, A. S. Schulz, D.B. Shmoys, C. Stein, and J. Wein. Improved approximation algorithms for minsum criteria. In *Proceedings of the 1996 International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science 1099*, Berlin, 1996. Springer-Verlag.

[3] Chris Coulston and Piotr Berman. Speed is more powerful than clairvoyance. In *Proceedings of the 6th Scandinavian Workshop on Algorithm Theory*, 1998.

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

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

[6] M. Goemans. A supermodular relaxation for scheduling with release dates. In *ipco96*, pages 288–300, 1996.

[7] 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.

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

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

[10] L. A. Hall, A. S. Schulz, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time: Off-line and on-line approximation algorithms. Submitted to Mathematics of Operations Research, 1996.

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

[12] Bala Kalyanasundaram and K. Pruhs. Speed is equivalent to clairvoyance. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 115–124, 1995.

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

[14] Bala Kalyanasundaram and K. Pruhs. Eliminating migration in multi-processor real-time scheduling. In *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms*, 1999.

[15] 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 26th Annual ACM Symposium on Theory of Computing*, May 1995.

[16] 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.

[17] 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.

[18] Tak Wah Lam and Kar Keung To. Trade-offs between speed and processor in hard-deadline scheduling. In *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms*, 1999.

[19] S. Leonardi and D. Raz. personal communication.

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

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

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

[23] C. Phillips, C. Stein, and J. Wein. Scheduling jobs that arrive over time. In *Proceedings of Fourth Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science, 955*, pages 86–97, Berlin, 1995. Springer-Verlag. Journal version to appear in Mathematical Programming B.

[24] M. Queyranne and A.S. Schulz. Polyhedral approaches to machine scheduling. Technical Report 408/1994, Technical University of Berlin, 1994.

[25] 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.

[26] D. B. Shmoys, J. Wein, and D.P. Williamson. Scheduling parallel machines on-line. *SIAM Journal on Computing*, 24:1313–1331, December 1995.