Noname manuscript No.
(will be inserted by the editor)

A Competitive Algorithm for Throughput Maximization on Identical Machines [⋆]

Benjamin Moseley · Kirk Pruhs · Clifford Stein · Rudy Zhou

Received: date / Accepted: date

Abstract This paper considers the basic problem of scheduling jobs online with preemption to maximize the number of jobs completed by their deadline on m identical machines. The main result is an $O(1)$ competitive deterministic algorithm for any number of machines $m > 1$.

Keywords Scheduling · Competitive Analysis · Online Algorithm

Mathematics Subject Classification (2020) 68W25 · 68W40

[⋆] previously appeared as an extended abstract in the conferece IPCO 2022 [12]

B. Moseley and R. Zhou were supported in part by NSF grants CCF-1824303, CCF-1845146, CCF-1733873 and CMMI-1938909. Benjamin Moseley was additionally supported in part by a Google Research Award, an Infor Research Award, and a Carnegie Bosch Junior Faculty Chair. K. Pruhs was supported in part by NSF grants CCF-1907673, CCF-2036077 and an IBM Faculty Award. C. Stein was partly supported by NSF Grants CCF-1714818 and CCF-1822809.

B. Moseley
Tepper School of Business, Carnegie Mellon University
E-mail: moseleyb@andrew.cmu.edu

K. Pruhs
Computer Science Department, University of Pittsburgh
E-mail: kirk@cs.pitt.edu

C. Stein
Industrial Engineering and Operations Research, Columbia University
E-mail: cliff@ieor.columbia.edu

Corresponding Author: R. Zhou
Tepper School of Business, Carnegie Mellon University
E-mail: rbz@andrew.cmu.edu

1 Introduction

We consider the basic problem of preemptively scheduling jobs that arrive online with sizes and deadlines on m identical machines to maximize the number of jobs that complete by their deadline.

Definition 1 (Throughput Maximization) Let J be a collection of jobs such that each $j \in J$ has a *release time* r_j, a *processing time* (or *size*) x_j, and a *deadline* d_j. The jobs arrive online at their release times, at which the scheduler becomes aware of job j and its x_j and d_j.

At each moment of time, the scheduler can specify up to m released jobs to run at this time, and the remaining processing time of the jobs that are run is decreased at a unit rate (so we assume that the online scheduler is allowed to produce a migratory schedule.) A job is completed if its remaining processing time drops to zero by the deadline of that job. The objective is to maximize the number completed jobs.

A key concept in designing algorithms for this problem is the laxity of a job. The *laxity* of a job j is $\ell_j = (d_j - r_j) - x_j$, which is the maximum amount of time we can not run j and still possibly complete it.

We measure the performance of our algorithm by the *competitive ratio*, which is the maximum over all instances of the ratio of the objective value of our algorithm to the objective value of the optimal offline schedule that is aware of all jobs in advance.

This problem is well understood for the $m = 1$ machine case. No $O(1)$-competitive deterministic algorithm is possible [2], but there is a randomized algorithm that is $O(1)$-competitive against an oblivious adversary [9], and there is a scalable ($O(1 + \epsilon)$-speed $O(1/\epsilon)$-competitive) deterministic algorithm [7]. The scalability result in [7] was extended to the case of $m > 1$ machines in [11].

Whether an $O(1)$-competitive algorithm exists for $m > 1$ machines has been open for twenty years. Previous results for the multiple machines setting require resource augmentation or assume that all jobs have high laxity [11, 5].

The main issue issue in removing these assumptions is how to determine which machine to assign a job to. If an online algorithm could determine which machine each job was assigned to in Opt, we could obtain an $O(1)$-competitive algorithm for $m > 1$ machines by a relatively straight-forward adaptation of the results from [9]. However, if the online algorithm ends up assigning some jobs to different machines than Opt, then comparing the number of completed jobs is challenging. Further, if jobs have small laxity, then the algorithm can be severely penalized for small mistakes in this assignment. One way to view the speed augmentation (or high laxity assumption) analyses in [11, 5] is that the speed augmentation assumption allows one to avoid having to address this issue in the analyses.

1.1 Our Results

Our main result is an $O(1)$-competitive deterministic algorithm for Throughput Maximization on $m > 1$ machines.

Theorem 1. *For all $m > 1$, there exists a deterministic $O(1)$-competitive algorithm for Throughput Maximization on m machines.*

We summarize our results and prior work in Table 1. Interestingly, notice that on a single machine there is no constant competitive deterministic algorithm, yet a randomized algorithm exists with constant competitive ratio. Our work shows that once more than one machine is considered, then determinism is sufficient to get a $O(1)$-competitive online algorithm.

	Deterministic	Randomized	Speed Augmentation
$m = 1$	$\omega(1)$ [2]	$O(1)$ [9]	$O(1+\epsilon)$-speed $O(1/\epsilon)$-competitive [7]
$m > 1$	$O(1)$ [This paper]	$O(1)$ [This paper]	$O(1+\epsilon)$-speed $O(1/\epsilon)$-competitive [11]

Table 1 Competitiveness Results

1.2 Scheduling Policies

We give some basic definitions and notations about scheduling policies.

A job j is *feasible* at time t (with respect to some schedule) if it can still be feasibly completed, so $x_j(t) > 0$ and $t + x_j(t) \leq d_j$, where $x_j(t)$ is the remaining processing time of job j at time t (with respect to the same schedule.)

Then a schedule $\mathcal{S}$ of jobs J is defined by a map from time/machine pairs (t, i) to a feasible job j that is run on machine i at time t, with the constraint that no job can be run one two different machines at the same time. . We conflate $\mathcal{S}$ with the scheduling policy as well as the set of jobs *completed* by the schedule. Thus, the objective value achieved by this schedule is $|\mathcal{S}|$.

A schedule is *non-migratory* if for every job j there exists a machine i such that if j is run at time t then j is run on machine i. Otherwise the schedule is *migratory.*

If $\mathcal{S}$ is a scheduling algorithm, then $\mathcal{S}(J, m)$ denotes the schedule that results from running $\mathcal{S}$ on instance J on m machines. Similarly, $\mathrm{Opt}(J, m)$ denotes the optimal schedule on instance J on m machines. We will sometimes omit the J and/or the m if they are clear from context. Sometimes we will abuse notation and let Opt denote a nearly-optimal schedule that additionally has some desirable structural property.

1.3 Algorithms and Technical Overview

A simple consequence of the results in [8] and [9] is an $O(1)$-competitive algorithm in the case that $m = O(1)$. Thus we concentrate on the case that m is large. Also note that since there is an $O(1)$-approximate non-migratory schedule [8], changing the number of machines by an $O(1)$ factor does not change the optimal objective value by more than an $O(1)$ factor. This is because we can always take an optimal non-migratory schedule on m machines and create a new schedule on m/c machines whose objective value decreases by at most a factor of c, by keeping the m/c machines that complete the most jobs.

These observations about the structure of near-optimal schedules allow us to design a $O(1)$-competitive algorithm that is a combination of various deterministic algorithms. In particular, on instance J our algorithm, FinalAlg will run a deterministic algorithm LMNY on $m/3$ machines on the subinstance $J_{hi} = \{j \in J \mid \ell_j > x_j\}$ of high laxity jobs, a deterministic algorithm SRPT on $m/3$ machines on the subinstance $J_{lo} = \{j \in J \mid \ell_j \le x_j\}$ of low laxity jobs, and a deterministic algorithm MLax on $m/3$ machines on the subinstance J_{lo} of low laxity jobs. Note that we run SRPT and MLax on the same jobs. To achieve this, if both algorithms decide to run the same job j, then the algorithm in which j has shorter remaining processing time actually runs job j, and the other simulates running j.

We will eventually show that for all instances, at least one of these three algorithms is $O(1)$-competitive, from which our main result will follow. Roughly, each of the three algorithms is responsible for a different part of Opt.

Our main theorem about FinalAlg is the following:

Theorem 2. *For any $m \ge 48$, FinalAlg is a $O(1)$-competitive deterministic algorithm for Throughput Maximization on m machines.*

We now discuss these three component algorithms of FinalAlg.

1.3.1 LMNY

The algorithm LMNY is the algorithm from [11] with the following guarantee.

Lemma 3. *[11] For any number of machines m, and any job instance J,* LMNY *is an $O(1)$-competitive deterministic algorithm on the instance J_{hi}.*

1.3.2 SRPT

The algorithm SRPT is a variation of the standard shortest remaining processing time algorithm:

Definition 2 (SRPT) At each time, run the m feasible jobs with shortest remaining processing time. If there are less than m feasible jobs, then all feasible jobs are run.

We will show that SRPT is competitive with the low laxity jobs in that are not preempted in Opt.

1.3.3 MLax

The final, most challenging, component algorithm of FinalAlg is MLax, which intuitively we want to be competitive on low-laxity jobs in Opt that are preempted.

To better understand the challenge of achieving this, consider $m = 1$ and an instance of disagreeable jobs, which means that jobs with an earlier release time have a later deadline. Further, suppose all jobs but one in Opt is preempted and completed at a later time.

To be competitive, MLax must preempt almost all the jobs that it completes, but cannot afford to abandon too many jobs that it preempts. Because the jobs have low laxity, this can be challenging as it can only preempt each job for a small amount of time, and its hard to know which of the many options is the "right" job to preempt for. This issue was resolved in [9] for the case of $m = 1$ machine, but the issue gets more challenging when $m > 1$, because we also have to choose the "right" machine to assign a job.

We now describe the algorithm MLax. Let $\alpha = O(1)$ be a sufficiently large constant (chosen later.) MLax maintains m stacks (last-in-first-out data structures) of jobs (one per machine), $H_1, \ldots, H_m$. The stacks are initially empty. At all times, MLax runs the top job of stack H_i on machine i. We define the *frontier* F to be the set consisting of the top job of each stack (i.e. all currently running jobs.) It remains to describe how the H_i's are updated.

There are two types of events that cause MLax to update the H_i's: reaching a job's pseudo-release time (defined below) or completing a job.

Definition 3 (Viable Jobs and Pseudo-Release Time) The pseudo-release time (if it exists) $\tilde{r}_j$ of job j is the earliest time in $[r_j, r_j + \frac{\ell_j}{2}]$ such that there are at least $\frac{7}{8}m$ jobs j' on the frontier satisfying $\alpha x_{j'} \geq \ell_j$.

We say a job j is *viable* if $\tilde{r}_j$ exists and *non-viable* otherwise.

At job j's pseudo-release time (note $\tilde{r}_j$ can be determined online by MLax), MLax does the following:

a) If there exists a stack whose top job j' satisfies $\alpha x_j \leq \ell_{j'}$, then *push* j onto any such stack.
b) Else if there exist at least $\frac{3}{4}m$ stacks whose second-top job j'' satisfies $\alpha x_j \leq \ell_{j''}$ and further some such stack has top job j' satisfying $\ell_j > \ell_{j'}$, then on such a stack with minimum $\ell_{j'}$, *replace* its top job j' by j.

While the replacement operation in step b can be implemented as a pop and then push, we view it as a separate operation for analysis purposes. To handle corner cases in these descriptions, one can assume that there is a job with infinite size/laxity on the bottom of each H_i.

When MLax completes a job j that was on stack H_i, MLax does the following:

c) *Pop* j off of stack H_i.
d) Keep *popping* H_i until the top job of H_i is feasible.

1.3.4 Analysis Sketch

There are three main steps in proving Theorem 2 to show FinalAlg is $O(1)$-competitive:

- In Section 2, we show how to modify the optimal schedule to obtain certain structural properties that facilitate the comparison with SRPT and MLax.
- In Section 3, we show that SRPT is competitive with the low-laxity, non-viable jobs. Intuitively, the jobs that MLax is running that prevent a job j from becoming viable are so much smaller than job j, and they provide a witness that SRPT must also be working on jobs much smaller than j.
- In Section 4, we show that SRPT and MLax together are competitive with the low-laxity, viable jobs. First, we show that SRPT is competitive with the number of non-preempted jobs in Opt. We then essentially show that MLax is competitive with the number of preempted jobs in Opt. The key component is the design of MLax is the condition that a job j won't replace a job on the frontier unless at there are at least $\frac{3}{4}m$ stacks whose second-top job j'' satisfies $\alpha x_j \leq \ell_{j''}$. This is the condition that intuitively most differentiates the MLax from m copies of the Lax algorithm in [9]. This also is the condition that allows us to surmount the issue of potentially assigning a job to a "wrong" processor. Jobs that satisfy this condition are highly flexible about where they can go on the frontier. Morally, our analysis shows that a constant fraction of the jobs that Opt preempts and completes must be such flexible jobs.

We combine these results in Section 5 to complete the analysis of FinalAlg.

1.4 Related Work

There is a line of papers that consider a dual version of the problem, where there is a constraint that all jobs must be completed by their deadline, and the objective is to minimize the number of machines used [13,4,1,6]. The current best known bound on the competitive ratio for this version is $O(\log \log m)$ from [6].

The speed augmentation results in [7,11] for throughput can be generalized to weighted throughput, where there a profit for each job, and the objective is to maximize the aggregate profit of jobs completed by their deadline. But without speed augmentation, $O(1)$-approximation is not possible for weighted throughput for any m, even allowing randomization [10].

There is also a line of papers that consider variations on online throughput scheduling in which the online scheduler has to commit to completing jobs at some point in time, with there being different variations of when commitment is required [11,5,3]. For example, [5] showed that there is a scalable algorithm for online throughput maximization that commits to finishing every job that it begins executing.

2 Structure of Optimal Schedule

The goal of this section is to introduce the key properties of (near-)optimal scheduling policies that we will use in our analysis.

For completeness, we show that by losing a constant factor in the competitive ratio, we can use a constant factor fewer machines than Opt. This justifies FinalAlg running each of three algorithms on $\frac{m}{3}$ machines.

Lemma 4. *For any collection of jobs J, number of machines m, and $c > 1$, we have $|\text{Opt}(J, \frac{m}{c})| = \Omega(\frac{1}{c}|\text{Opt}(J, m)|)$.*

Proof. It is shown in [8] that for any schedule on m machines, there exists a non-migratory schedule on at most $6m$ machines that completes the same jobs. Applied to $\text{Opt}(J, m)$, we obtain a non-migratory schedule $\mathcal{S}$ on $6m$ machines with $|\mathcal{S}| = |\text{Opt}(J, m)|$. Keeping the $\frac{m}{c}$ machines that complete the most jobs in $\mathcal{S}$ gives a non-migratory schedule on $\frac{m}{c}$ machines that completes at least $\frac{1}{6c}|\mathcal{S}|$ jobs. □

A non-migratory schedule on m machines can be expressed as m schedules, each on a single machine and a separate set of jobs. To characterize these single machine schedules, we introduce the concept of forest schedules. Let $\mathcal{S}$ be any schedule. For any job j, we let $f_j(\mathcal{S})$ and $c_j(\mathcal{S})$ denote the first and last times that $\mathcal{S}$ runs the job j, respectively. Note that $\mathcal{S}$ does not necessarily complete j at time $c_j(\mathcal{S})$.

Definition 4 (Forest Schedule) We say a single-machine schedule $\mathcal{S}$ is a forest schedule if for all jobs j, j' such that $f_j(\mathcal{S}) < f_{j'}(\mathcal{S})$, then $\mathcal{S}$ does not run j during the time interval $(f_{j'}(\mathcal{S}), c_{j'}(\mathcal{S}))$ (so the $(f_j(\mathcal{S}), c_j(\mathcal{S}))$-intervals form a laminar family.) Then $\mathcal{S}$ naturally defines a forest (in the graph-theoretic sense), where the nodes are jobs run by $\mathcal{S}$ and the descendants of a job j are the the jobs that are first run in the time interval $(f_j(\mathcal{S}), c_j(\mathcal{S}))$.

Then a non-migratory m-machine schedule is a forest schedule if all of its single-machine schedules are forest schedules.

With these definitions, we are ready to construct the near-optimal policies that we will compare SRPT and MLax to:

Lemma 5. *Let J be a set of jobs satisfying $\ell_j \le x_j$ for all $j \in J$. Then for any times $\hat{r}_j \in [r_j, r_j + \frac{\ell_j}{2}]$ and constant $\alpha \ge 1$, there exist non-migratory forest schedules $\mathcal{S}$ and $\mathcal{S}'$ on the jobs J such that:*

1. *Both $\mathcal{S}$ and $\mathcal{S}'$ complete every job they run.*
2. *Let J_i be the set of jobs that $\mathcal{S}$ runs on machine i. For every machine i and time, if there exists a feasible job in J_i, then $\mathcal{S}$ runs such a job.*
3. *For all jobs $j \in \mathcal{S}$, we have $f_j(\mathcal{S}) = \hat{r}_j$.*
4. *If job j' is a descendant of job j in $\mathcal{S}$, then $\alpha x_{j'} \le \ell_j$*
5. *$|\{\text{leaves of } \mathcal{S}'\}| + |\mathcal{S}| = \Omega(|\text{Opt}(J)|)$.*

Proof. We modify the optimal schedule $\text{Opt}(J)$ to obtain the desired properties. First, we may assume that $\text{Opt}(J)$ is non-migratory by losing a constant factor (Lemma 4.) Thus, it suffices to prove the lemma for a single machine schedule, because we can apply the lemma to each of the single-machine schedules in the non-migratory schedule $\text{Opt}(J)$. The proof for the single-machine case follows from the modifications given in Lemmas 22 and 23 of [9]. We note that [9] only show how to ensure $f_j(\mathcal{S}) = t_j$ for a particular $t_j \in [r_j, r_j + \frac{\ell_j}{2}]$, but it is straightforward to verify that the same proof holds for any $t_j \in [r_j, r_j + \frac{\ell_j}{2}]$. □

Morally, the schedule $\mathcal{S}$ captures the jobs in Opt that are preempted and $\mathcal{S}'$ captures the jobs in Opt that are not preempted (i.e. the leaves in the forest schedule.)

3 SRPT is Competitive with Non-Viable Jobs

The main result of this section is that SRPT is competitive with the number of non-viable, low-laxity jobs of the optimal schedule (Theorem 6.) We recall that a job j is non-viable if for *every time* in $[r_j, r_j + \frac{\ell_j}{2}]$, there are at least $\frac{1}{8}m$ jobs j' on the frontier of MLax satisfying $\alpha x_{j'} < \ell_j$.

Theorem 6. *Let J be a set of jobs satisfying $\ell_j \leq x_j$ for all $j \in J$. Then for $\alpha = O(1)$ sufficiently large and number of machines $m \geq 16$, we have $|\text{SRPT}(J)| = \Omega(|\text{Opt}(J_{nv})|)$, where J_{nv} is the set of non-viable jobs with respect to* MLax(J).

In the remainder of this section, we prove Theorem 6. The main idea of the proof is that for any non-viable job j, MLax is running many jobs that are much smaller than j (by at least an α-factor.) These jobs give a witness that SRPT must be working on these jobs or even smaller ones.

We begin with a lemma stating that SRPT is competitive with the leaves of any forest schedule. Intuitively this follows because whenever some schedule is running a feasible job, then SRPT either runs the same job or a job with shorter remaining processing time. We will use this lemma to handle the non-viable jobs that are not preempted.

Lemma 7. *Let J be any set of jobs and $\mathcal{S}$ be any forest schedule on m machines and jobs $J' \subset J$ that only runs feasible jobs. Let L be the set of leaves of $\mathcal{S}$. Then $|\text{SRPT}(J)| \geq \frac{1}{2}|L|$*

Proof. It suffices to show that $|L \setminus \text{SRPT}(J)| \leq |\text{SRPT}(J)|$. The main property of SRPT gives:

Proposition 8. *Consider any leaf $\ell \in L \setminus \text{SRPT}(J)$. Suppose $\mathcal{S}$ starts running ℓ at time t. Then* SRPT *completes m jobs in the time interval $[f_\ell(\mathcal{S}), f_\ell(\mathcal{S}) + x_\ell]$.*

Proof. At time $f_\ell(\mathcal{S})$ in SRPT (J), job ℓ has remaining processing time at most x_ℓ and is feasible by assumption. Because $\ell \notin \mathrm{SRPT}(J)$, there must exist a first time $t' \in [f_\ell(\mathcal{S}), f_\ell(\mathcal{S}) + x_\ell]$ where ℓ is not run by $\mathrm{SRPT}(J)$. At this time, $\mathrm{SRPT}(J)$ must be running m jobs with remaining processing time at most $x_\ell - (t' - f_\ell(\mathcal{S}))$. In particular, $\mathrm{SRPT}(J)$ must complete m jobs by time $f_\ell(\mathcal{S}) + x_\ell$. □

Using the proposition, we give a charging scheme: Each job $\ell \in L \setminus \mathrm{SRPT}(J)$ begins with 1 credit. By the proposition, we can find m jobs that $\mathrm{SRPT}(J)$ completes in the time interval $[f_\ell(\mathcal{S}), f_\ell(\mathcal{S}) + x_\ell]$. Then ℓ transfers $\frac{1}{m}$ credits each to m such jobs in SRPT.

It remains to show that each $j \in \mathrm{SRPT}(J)$ gets at most 1 credit. Note that j can only get credits from leaves ℓ such that $c_j(\mathrm{SRPT}) \in [f_\ell(\mathcal{S}), f_\ell(\mathcal{S}) + x_\ell]$. There are at most m such intervals (at most one per machine), because we only consider leaves, whose intervals are disjoint if there are on the same machine. □

Now we are ready to prove Theorem 6.

Proof of Theorem 6. Let $\mathcal{S}, \mathcal{S}'$ be the schedules guaranteed by Lemma 5 on the set of jobs J_{nv} with $\hat{r}_j = r_j$ for all $j \in J_{nv}$. We re-state the properties of these schedules for convenience:

1. Both $\mathcal{S}$ and $\mathcal{S}'$ complete every job they run.
2. Let J_i be the set of jobs that $\mathcal{S}$ runs on machine i. For every machine i and time, if there exists a feasible job in J_i, then $\mathcal{S}$ runs such a job.
3. For all jobs $j \in \mathcal{S}$, we have $f_j(\mathcal{S}) = r_j$.
4. If job j' is a descendant of job j in $\mathcal{S}$, then $\alpha x_{j'} \leq \ell_j$
5. $|\{\text{leaves of } \mathcal{S}'\}| + |\mathcal{S}| = \Omega(|\mathrm{Opt}(J_{nv})|)$.

By Lemma 7, we have $|\mathrm{SRPT}(J)| = \Omega(|\{\text{leaves of } \mathcal{S}'\}|)$. Thus, it remains to show the following:

Lemma 9. *For $\alpha = O(1)$ sufficiently large, $|\mathrm{SRPT}(J)| = \Omega(|\mathcal{S}|)$*

Proof. We first show that for the majority of jobs j in $\mathcal{S}$'s forest, we run j itself on some machine for at least a constant fraction of the time interval $[r_j, r_j + \frac{\ell_j}{2}]$.

Proposition 10. *For at least half of the nodes j in $\mathcal{S}$'s forest, there exists a closed interval $I_j \subset [r_j, r_j + \frac{\ell_j}{2}]$ of length at least $\frac{\ell_j}{8}$ such that $\mathcal{S}$ runs j on some machine during I_j.*

Proof. We say a node j is a *non-progenitor* if j has less than 2^z descendants at depth z from j for all $z \geq 1$. Because $\mathcal{S}$ satisfies (1), at least half of the nodes in $\mathcal{S}$'s forest are non-progenitors. This follows from Lemma 7 in [9].

Now consider any non-progenitor node j. Because $\mathcal{S}$ is a forest, $\mathcal{S}$ is only running j or its descendants on some machine in times $[r_j, r_j + \frac{\ell_j}{2}]$. Further, because j is a non-progenitor and $\mathcal{S}$ satisfies (1) and (4), we can partition

$[r_j, r_j + \frac{\ell_j}{2}] = [r_j, a] \cup (a, b) \cup [b, r_j + \frac{\ell_j}{2}]$ such that $[r_j, a]$ and $[b, r_j + \frac{\ell_j}{2}]$ are times where $\mathcal{S}$ is running j, and (a, b) are times where $\mathcal{S}$ is running descendants of j. By taking α sufficiently large, we have $|(a, b)| \leq \frac{\ell_j}{4}$. This follows from Lemma 6 in [9]. It follows, at least one of $[r_j, a]$ or $[b, r_j + \frac{\ell_j}{2}]$ has length at least $\frac{\ell_j}{8}$. This gives the desired I_j. □

Let $\mathcal{S}'' \subset \mathcal{S}$ be the collection of jobs guaranteed by the proposition, so $|\mathcal{S}''| \geq \frac{1}{2}|\mathcal{S}|$. It suffices to show that $|\mathrm{SRPT}(J)| = \Omega(|\mathcal{S}''|)$. Thus, we argue about $\mathrm{ML{\small AX}}(J)$ in the interval I_j (guaranteed by Proposition 10) for some $j \in \mathcal{S}''$.

Proposition 11. *Consider any job $j \in \mathcal{S}''$. For sufficiently large $\alpha = O(1)$, $\mathrm{ML{\small AX}}(J)$ starts running at least $\frac{m}{16}$ jobs during I_j such that each such job j' satisfies $[f_{j'}(\mathrm{ML{\small AX}}(J)), f_{j'}(\mathrm{ML{\small AX}}(J)) + x_{j'}] \subset I_j$.*

Proof. We let I' be the prefix of I_j with length exactly $\frac{|I_j|}{2} \geq \frac{\ell_j}{16}$. Recall that $j \in \mathcal{S}''$ is non-viable. Thus, because $I' \subset I_j \subset [r_j, r_j + \frac{\ell_j}{2}]$, $\mathrm{ML{\small AX}}(J)$ is always running at least $\frac{1}{8}m$ jobs j' satisfying $\alpha x_{j'} < \ell_j$ during I'.

We define J' to be the set of jobs that $\mathrm{ML{\small AX}}(J)$ runs during I' satisfying $\alpha x_{j'} < \ell_j$. We further partition J' into size classes, $J' = \bigcup_{z \in \mathbb{N}} J'_z$ such that J'_z consists of the jobs in J' with size in $(\frac{\ell_j}{\alpha^{z+1}}, \frac{\ell_j}{\alpha^z}]$.

For each machine i, we let T^i_z be the times in I' that $\mathrm{ML{\small AX}}(J)$ is running a job from J'_z on machine i. Note that each T^i_z is the union of finitely many intervals. Then because $\mathrm{ML{\small AX}}(J)$ is always running at least $\frac{1}{8}m$ jobs j' satisfying $\alpha x_{j'} \leq \ell_j$ during I', we have:

$$\sum_{z \in \mathbb{N}} \sum_{i \in [m]} |T^i_z| \geq \frac{m}{8}|I'|.$$

By averaging, there exists some z with $\sum_{i \in [m]} |T^i_z| \geq \frac{m}{8} \frac{|I'|}{2^{z+1}}$.

Fix such a z. It suffices to show that there exist at least $\frac{m}{16}$ jobs in J'_z that $\mathrm{L{\small AX}}$ starts in I'. This is because every job $j' \in J'_z$ has size at most $\frac{\ell_j}{\alpha^z}$ and $|I_j \setminus I'| \geq \frac{\ell_j}{16}$. Taking $\alpha \geq 16$ gives that $f_{j'}(\mathrm{L{\small AX}}) + x_{j'} \in I_j$.

Note that every job in J'_z has size within a α-factor of each other, so there can be at most one such job per stack at any time. This implies that there are at most m jobs in J'_z that *don't* start in I' (i.e. the start before I'.) These jobs contribute at most $m\frac{\ell_j}{\alpha^z}$ to $\sum_{i \in [m]} |T^i_z|$. Choosing α large enough, we can ensure that the jobs in J'_z that start in I' contribute at least $\frac{m}{16}\frac{\ell_j}{\alpha^z}$ to $\sum_{i \in [m]} |T^i_z|$. To conclude, we note that each job in J'_z that starts in I' contributes at most $\frac{\ell_j}{\alpha^z}$ to the same sum, so there must exist at least $\frac{m}{16}$ such jobs. □

Using the above proposition, we define a charging scheme to show that $|\mathrm{SRPT}(J)| = \Omega(|\mathcal{S}''|)$. Each job $j \in \mathcal{S}''$ begins with 1 credit. By the proposition, we can find $\frac{m}{16}$ jobs j' such that $[f_{j'}(\mathrm{ML{\small AX}}(J)), f_{j'}(\mathrm{ML{\small AX}}(J)) + x_{j'}]$ is contained in the time when $\mathcal{S}$ runs j. There are two cases to consider. If all

$\frac{m}{16}$ jobs we find are contained in SRPT(J), then we transfer $\frac{16}{m}$ credits from j to each of the $\frac{m}{16}$-many jobs. Note that here we are using $m \geq 16$. Otherwise, there exists some such j' that is not in SRPT(J). Then SRPT(J) will complete at least m jobs in $[f_{j'}(\text{MLax}(J)), f_{j'}(\text{MLax}(J)) + x_{j'}]$. We transfer $\frac{1}{m}$ credits from j to each of the m-many jobs. To conclude, we note that each $j' \in$ SRPT(J) gets $O(\frac{1}{m})$ credits from at most $O(m)$ jobs in $\mathcal{S}''$. □

□

4 SRPT and MLax are Competitive with Viable Jobs

We have shown that SRPT is competitive with the non-viable, low-laxity jobs. Thus, it remains to account for the viable, low-laxity jobs. We recall that a job j is viable if there exists a time in $[r_j, r_j + \frac{\ell_j}{2}]$ such that there are at least $\frac{7}{8}m$ jobs j' on the frontier satisfying $\alpha x_{j'} \geq \ell_j$. The first such time is the pseudo-release time, $\tilde{r}_j$ of job j. For these jobs, we show that SRPT and MLax together are competitive with the viable, low-laxity jobs of the optimal schedule.

Theorem 12. *Let J be a set of jobs satisfying $\ell_j \leq x_j$ for all $j \in J$. Then for $\alpha = O(1)$ sufficiently large and number of machines $m \geq 8$, we have $|\text{SRPT}(J)| + |\text{MLax}(J)| = \Omega(|\text{Opt}(J_v)|)$, where J_v is the set of viable jobs with respect to* MLax(J).

Proof of Theorem 12. Let $\mathcal{S}, \mathcal{S}'$ be the schedules guaranteed by Lemma 5 on the set of jobs J_v with $\hat{r}_j = \tilde{r}_j$ for all $j \in J_v$. We re-state the properties of these schedules for convenience:

1. Both $\mathcal{S}$ and $\mathcal{S}'$ complete every job they run.
2. Let J_i be the set of jobs that $\mathcal{S}$ runs on machine i. For every machine i and time, if there exists a feasible job in J_i, then $\mathcal{S}$ runs such a job.
3. For all jobs $j \in \mathcal{S}$, we have $f_j(\mathcal{S}) = \tilde{r}_j$.
4. If job j' is a descendant of job j in $\mathcal{S}$, then $\alpha x_{j'} \leq \ell_j$
5. $|\{\text{leaves of } \mathcal{S}'\}| + |\mathcal{S}| = \Omega(|\text{Opt}(J_v)|)$.

By Lemma 7, we have $|\text{SRPT}(J)| = \Omega(|\{\text{leaves of } \mathcal{S}'\}|)$. Thus, it suffices to show that $|\text{SRPT}(J)| + |\text{MLax}(J)| = \Omega(|\mathcal{S}|)$. We do this with two lemmas, whose proofs we defer until later. First, we show that MLax pushes (not necessarily completes) many jobs. In particular, we show:

Lemma 13. $|\text{SRPT}(J)| + \#(\textit{pushes of}\ \text{MLax}(J)) = \Omega(|\mathcal{S}|)$

The main idea to prove Lemma 13 is to consider sequences of preemptions in Opt. In particular, suppose Opt preempts job a for b and then b for c. Roughly, we use viability to show that the only way MLax doesn't push any of these jobs is if in between their pseudo-release times, MLax pushes $\Omega(m)$ jobs.

Second, we show that the pushes of MLax give a witness that SRPT and MLax together actually complete many jobs.

Lemma 14. $|\text{SRPT}(J)| + |\text{MLax}(J)| = \Omega(\#(\textit{pushes of}\ \text{MLax}(J)))$.

The main idea to prove Lemma 14 is to upper-bound the number of jobs that MLax pops because they are infeasible (all other pushes lead to completed jobs.) The reason MLax pops a job j for being infeasible is because while j was on a stack, MLax spent at least $\frac{\ell_j}{2}$ units of time running jobs higher than j on j's stack. Either those jobs are completed by MLax, or MLax must have have done many pushes or replacements instead. We show that the replacements give a witness that SRPT must complete many jobs.

Combining these two lemmas completes the proof of Theorem 12. □

Now we go back and prove Lemma 13 and Lemma 14.

4.1 Proof of Lemma 13

Recall that $\mathcal{S}$ is a forest schedule. We say the *first child* of a job j is the child j' of j with the earliest starting time $f_{j'}(\mathcal{S})$. In other words, if j is not a leaf, then its first child is the first job that pre-empts j. We first focus on a sequence of first children in $\mathcal{S}$.

Lemma 15. *Let $a, b, c \in \mathcal{S}$ be jobs such that b is the first child of a and c is the first child of b. Then* MLax(J) *does at least one of the following during the time interval $[\tilde{r}_a, \tilde{r}_c]$:*

- *Push at least $\frac{m}{8}$ jobs*
- *Push job b*
- *Push a job on top of b when b is on the frontier*
- *Push c*

Proof. By the properties of $\mathcal{S}$, we have $\tilde{r}_a < \tilde{r}_b < \tilde{r}_c$. It suffices to show that if during $[\tilde{r}_a, \tilde{r}_c]$, MLax$(J)$ pushes strictly fewer than $\frac{m}{8}$ jobs, MLax(J) does not push b, and MLax(J) does not push any job on top of b if b is on the frontier, then MLax(J) pushes c.

First, because MLax(J) pushes strictly fewer than $\frac{m}{8}$ jobs during $[\tilde{r}_a, \tilde{r}_c]$, there exists at least $\frac{7}{8}m$ stacks that receive no push during this interval. We call such stacks *stable*. The key property of stable stacks is that the laxities of their top- and second-top jobs never decrease during this interval, because these stacks are only changed by replacements and pops.

Now consider time $\tilde{r}_a$. By definition of pseudo-release time, at this time, there exist at least $\frac{7}{8}m$ stacks whose top job j' satisfies $\alpha x_{j'} \geq \ell_j$. Further, for any such stack, let j'' be its second-top job. Then because b is a descendant of a in $\mathcal{S}$, we have:

$$\alpha x_b \leq \ell_a \leq \alpha x_{j'} \leq \ell_{j''}.$$

It follows, there exist at least $\frac{3}{4}m$ stable stacks whose second-top job j'' satisfies $\alpha x_b \leq \ell_{j''}$ for the entirety of $[\tilde{r}_a, \tilde{r}_c]$. We say such stacks are b-stable.

Now consider time $\tilde{r}_b$. We may assume b is not pushed at this time. However, there exist at least $\frac{3}{4}m$ that are b-stable. Thus, if we do not replace the top of some stack with b, it must be the case that the top job j' of every b-stable stack satisfies $\ell'_j \geq \ell_b$. Because these stacks are stable, their laxities only increase by time $\tilde{r}_c$, so MLax(J) will push c on some stack at that time.

Otherwise, suppose we replace the top job of some stack with b. In particular, b is on the frontier at $\tilde{r}_b$. We may assume that no job is pushed directly on top of b. If b remains on the frontier by time $\tilde{r}_c$, then MLax(J) will push c, because $\alpha x_c \leq \ell_b$. The remaining case is if b leaves the frontier in some time in $[\tilde{r}_b, \tilde{r}_c]$. We claim that it cannot be the case that b is popped, because by (2), $\mathcal{S}$ could not complete b by time $\tilde{r}_c$, so MLax(J) cannot as well. Thus, it must be the case that b is replaced by some job, say d at time $\tilde{r}_d$. At this time, there exist at least $\frac{3}{4}m$ stacks whose second-top job j'' satisfies $\alpha x_d \leq \ell_{j''}$. It follows, there exist at least $\frac{m}{2}$ b-stable stacks whose second-top job j'' satisfies $\alpha x_d \leq \ell_{j''}$ at time $\tilde{r}_d$. Note that because $m \geq 8$, there exists at least one such stack, say i, that is not b's stack. In particular, because b's stack has minimum laxity, it must be the case that the top job j' of stack i satisfies $\ell_{j'} \geq \ell_b$. Finally, because stack i is stable, at time $\tilde{r}_c$ we will push c. □

Now using the above lemma, we give a charging scheme to prove Lemma 13. First note that by Lemma 7, we have $|\text{SRPT}(J)| = \Omega(\#(\text{leaves of } \mathcal{S}))$. Thus, it suffices to give a charging scheme such that each job $a \in \mathcal{S}$ begins with 1 credit, and charges it to leaves of $\mathcal{S}$ and completions of MLax(J) so that each job is charged at most $O(1)$ credits. Each job $a \in \mathcal{S}$ distributes its 1 credit as follows:

– (Leaf Transfer) If a is a leaf or parent of a leaf of $\mathcal{S}$, say ℓ, then a charges ℓ for 1 credit.

Else let b be the first child of a and c the first child of b in $\mathcal{S}$

– (Push Transfer) If MLax(J) pushes b or c, then a charges 1 unit to b or c, respectively.
– (Interior Transfer) Else if job b is on the frontier, but another job, say d, is pushed on top of b, then a charges 1 unit to d.
– (m-Push Transfer) Otherwise, by Lemma 15, MLax(J) must push at least $\frac{m}{8}$ jobs during $[\tilde{r}_a, \tilde{r}_c]$. In this case, a charges $\frac{8}{m}$ units to each of these $\frac{m}{8}$ such jobs.

This completes the description of the charging scheme. It remains to show that each job is charged at most $O(1)$ credits. Each job receives at most 2 credits due to Leaf Transfers and at most 2 credits due to Push Transfers and Interior Transfers. As each job is in at most $3m$ intervals of the form $[\tilde{r}_a, \tilde{r}_c]$, each job is charged $O(1)$ from m-Push Transfers.

4.2 Proof of Lemma 14

Recall in MLax, there are two types of pops: a job is popped if it is completed, and then we continue popping until the top job of that stack is feasible. We call the former *completion pops* and the later *infeasible pops*. Note that it suffices to prove the next lemma, which bounds the infeasible pops. This is because #(pushes of MLax(J)) = #(completions pops of MLax(J))+#(infeasible pops of MLax(J)). To see this, note that every stack is empty at the beginning and end of the algorithm, and the stack size only changes due to pushes and pops.

Lemma 16. *For $\alpha = O(1)$ sufficiently large, we have:*

$|\mathrm{SRPT}(J)|+|\mathrm{MLax}(J)|+\#($*pushes of* $\mathrm{MLax}(J)) \geq 2\cdot\#($*infeasible pops of* $\mathrm{MLax}(J))$.

Proof. We define a charging scheme such that the completions of SRPT(J) and MLax(J) and the pushes executed by MLax(J) pay for the infeasible pops. Each completion of SRPT(J) is given 2 credits, each completion of MLax(J) is given 1 credit, and each job that MLax(J) pushes is given 1 credit. Thus each job begins with at most 4 credits. For any $z \geq 0$, we say job j' is z-below j (at time t) if j' and j are on the same stack in MLax(J) and j' is z positions below j on that stack at time t. We define z-above analogously. A job j distributes these initial credits as follows:

– (SRPT-transfer) If SRPT(J) completes job j and MLax also ran j at some point, then j gives $\frac{1}{2^{z+1}}$ credits to the job that is z-below j at time $f_j(\mathrm{MLax}(J))$ for all $z \geq 0$.
– (m-SRPT-transfer) If SRPT(J) completes job j at time t, then j gives $\frac{1}{2^{z+1}}\frac{1}{m}$ credits to the job that is z-below the top of each stack in MLax(J) at time t for all $z \geq 0$.
– (MLax-transfer) If MLax(J) completes a job j, then j gives $\frac{1}{2^{z+1}}$ credits to the job that is z-below j at the time j is completed for all $z \geq 0$.
– (Push-transfer) If MLax(J) pushes a job j, then j gives $\frac{1}{2^{z+1}}$ credits to the job that is z-below j at the time j is pushed for all $z \geq 0$.

It remains to show that for $\alpha = O(1)$ sufficiently large, every infeasible pop gets at least 4 credits. We consider any job j that is an infeasible pop of MLax(J). At time $\tilde{r}_j$ when j joins some stack in MLax(J), say H, j's remaining laxity was at least $\frac{\ell_j}{2}$. However, as j later became an infeasible pop, it must be the case that while j was on stack H, MLax(J) was running jobs that are higher than j on stack H for at least $\frac{\ell_j}{2}$ units of time.

Let I be the union of intervals of times that MLax(J) runs a job higher than j on stack H (so j is on the stack for the entirety of I.) Then we have $|I| \geq \frac{\ell_j}{2}$. Further, we partition I based on the height of the job on H that MLax(J) is currently running. In particular, we partition $I = \bigcup_{z\geq 1} I_z$, where I_z is the union of intervals of times that MLax(J) runs a job on H that is exactly z-above j.

By averaging, there exists a $z \geq 1$ such that $|I_z| \geq \frac{\ell_j}{2^{z+1}}$. Fix such a z. We can write I_z as the union of disjoint intervals, say $I_z = \bigcup_{u=1}^{s}[a_u, b_u]$. Because during each sub-interval, MLax(J) is running jobs on H that are much smaller than j itself, these jobs give a witness that SRPT(J) completes many jobs as long as these sub-intervals are long enough. We formalize this in the following proposition.

Proposition 17. *In each sub-interval* $[a_u, b_u]$ *of length at least* $4\frac{\ell_j}{\alpha^z}$, *job* j *earns at least* $\frac{1}{2^{z+3}} \frac{b_u - a_u}{\ell_j/\alpha^z}$ *credits from* SRPT*-transfers and* m-SRPT*-transfers.*

Proof. Because $[a_u, b_u]$ has length at least $4\frac{\ell_j}{\alpha^z}$, we can partition $[a_u, b_u]$ into sub-sub-intervals such that all but at most one sub-sub-interval has length exactly $2\frac{\ell_j}{\alpha^z}$. In particular, we have at least $\frac{1}{2}\frac{b_u - a_u}{2\ell_j/\alpha^z}$ sub-sub intervals of length exactly $2\frac{\ell_j}{\alpha^z}$.

Now consider any such sub-sub-interval. During this time, MLax(J) only runs jobs on H that are z-above j. Let J_z be the set of z-above jobs that MLax(J) runs during I_z. For every job $j' \in J_z$, we have $x_{j'} \leq \frac{\ell_j}{\alpha^z}$. It follows that j' is on stack H for at most $x_{j'} \leq \frac{\ell_j}{\alpha^z}$ units of time. In particular, MLax(J) must *start* a new z-above job, say j', in the first half of the sub-sub-interval at some time, say t.

At time t, j' is feasible. There are two cases to consider. If SRPT(J) also completes j' at some time, then j get $\frac{1}{2^{z+1}}$ credits from j' in a SRPT-transfer. Otherwise if SRPT(J) never completes j', then because j' is feasible at t, it must be the case that SRPT(J) completes m jobs during the sub-sub-interval. Thus, j gets $\frac{1}{m}\frac{1}{2^{z+1}}$ credits from m separate m-SRPT-transfers during this sub-sub-interval. We conclude, job j gets at least $\frac{1}{2^{z+1}}$ credits from at least $\frac{1}{2}\frac{b_u - a_u}{2\ell_j/\alpha^z}$ sub-sub-intervals. □

On the other hand, even if the sub-intervals are too short, the job j still gets credits from MLax-transfers and Push-transfers when the height of the stack changes. We formalize this in the following proposition.

Proposition 18. *For every sub-interval* $[a_u, b_u]$, *job* j *earns at least* $\frac{1}{2^{z+2}}$ *credits from* MLax*-transfers and Push-transfers at time* b_u.

Proof. Up until time b_u, MLax(J) was running a z-above job on stack H. At time b_u, the height of the stack H must change. If the height decreases, then it must be the case that MLax(J) completes the z-above job, so j will get $\frac{1}{2^{z+1}}$ credits from a MLax(J)-transfer. Otherwise, the height increases, so MLax(J) must push a job that is $z+1$-above j, which gives j $\frac{1}{2^{z+2}}$ credits from a Push-transfer. □

Now we combine the above two propositions to complete the proof of Lemma 16. We say a sub-interval $[a_u, b_u]$ is *long* if it has length at least $4\frac{\ell_j}{\alpha^z}$ (i.e. we can apply Proposition 17 to it) and *short* otherwise. First, suppose the aggregate length of all long intervals it at least $4 \cdot 2^{z+3}\frac{\ell_j}{\alpha^z}$. Then by Proposition 17, job j gets at least 4 credits from the long intervals. Otherwise, the aggregate length of all long intervals is less than $4 \cdot 2^{z+3}\frac{\ell_j}{\alpha^z}$. In this case, recall that the long and short intervals partition I_z, which has length at least $\frac{\ell_j}{2^{z+1}}$. It follows, the aggregate length of the short intervals is at least $\frac{\ell_j}{2^{z+1}} - 4 \cdot 2^{z+3}\frac{\ell_j}{\alpha^z}$. For $\alpha = O(1)$ large enough, we may assume the aggregate length of the short intervals is at least $4 \cdot 2^{z+2}\frac{4\ell_j}{\alpha^z}$. Because each short interval has length at most $4\frac{\ell_j}{\alpha^z}$, there are at least $4 \cdot 2^{z+2}$ short intervals. We conclude, by Proposition 18, job j gets at least 4 credits from the short intervals. We conclude, in either case job j gets at least 4 credits. □

5 Putting it all together

In this section, we prove our main result, Theorem 1, which follows from the next meta-theorem:

Theorem 19. *Let J be any set of jobs. Then for number of machines $m \geq 16$, we have $|\text{LMNY}(J_{hi})| + |\text{SRPT}(J_{lo})| + |\text{MLax}(J_{lo})| = \Omega(|\text{Opt}(J)|)$, where $J_{hi} = \{j \in J \mid \ell_j > x_j\}$ and $J_{lo} = \{j \in J \mid \ell_j \leq x_j\}$ partition J into high- and low-laxity jobs.*

Proof. We have $|\text{LMNY}(J_{hi})| = \Omega(|\text{Opt}(J_{hi})|$ by Lemma 3. Also, we further partition $J_{lo} = J_v \cup J_{nv}$ into the viable and non-viable jobs with respect to $\text{MLax}(J_{lo})$. Then Theorem 6 and Theorem 12 together give $|\text{SRPT}(J_{lo})| + |\text{MLax}(J_{lo})| = \Omega(|\text{Opt}(J_v)| + |\text{Opt}(J_{nv})|)$. To complete the proof, we observe that $J = J_{hi} \cup J_v \cup J_{nv}$ partitions J, so $|\text{Opt}(J_{hi})| + |\text{Opt}(J_v)| + |\text{Opt}(J_{nv})| = \Omega(|\text{Opt}(J)|)$. □

The proof of Theorem 2, which gives our performance guarantee for FinalAlg is immediate:

Proof of Theorem 2. By combining Theorem 19 and Lemma 4, the objective value achieved by FinalAlg is:

$$\begin{aligned}\Omega(|\text{LMNY}(J_{hi}, \frac{m}{3})| + |\text{SRPT}(J_{lo}, \frac{m}{3})| + |\text{MLax}(J_{lo}, \frac{m}{3})|) &= \Omega(|\text{Opt}(J, \frac{m}{3})|) \\ &= \Omega(|\text{Opt}(J, m)|).\end{aligned}$$

□

Finally, we obtain our $O(1)$-competitive deterministic algorithm for all $m > 1$ (recall FinalAlg is $O(1)$-competitive only when $m \geq 48$) by using a two-machine algorithm when m is too small:

Proof of Theorem 1. Our algorithm is the following: If $1 < m < 48$, then we run the deterministic two-machine algorithm from [9] which is $O(1)$-competitive with the optimal single-machine schedule. Thus by Lemma 4, this algorithm is also $O(m) = O(1)$-competitive for all $m < 48$. Otherwise, $m \geq 48$, so we run FinalAlg. □

6 Conclusion

We considered throughput maximization on multiple machines without any high-laxity or speed augmentation assumption. Our final competitive ratio is a large unspecified constant (obtained by choosing $\alpha = O(1)$ sufficiently large.) Many interesting open questions remain. What is the right competitive ratio for throughput maximization? Can we give an improved upper bound, or any lower bound? The only known lower bounds rule out constant-competitive deterministic algorithms for $m = 1$. Further, is migration necessary to achieve constant-competitiveness? Note that two out of three of our component algorithms (LMNY and LAX) are non-migratory. Naively, SRPT is migratory, and we also use migration because LAX and SRPT share the low-laxity jobs. We leave it as an open question to develop a *non-migratory* constant-competitive algorithm for throughput maximization on multiple machines.

References

1. Azar, Y., Cohen, S.: An improved algorithm for online machine minimization. Operations Research Letters **46**(1), 128–133 (2018)
2. Baruah, S.K., Koren, G., Mao, D., Mishra, B., Raghunathan, A., Rosier, L.E., Shasha, D.E., Wang, F.: On the competitiveness of on-line real-time task scheduling. Real Time Systems **4**(2), 125–144 (1992)
3. Chen, L., Eberle, F., Megow, N., Schewior, K., Stein, C.: A general framework for handling commitment in online throughput maximization. Mathematical Programming **183**(1), 215–247 (2020)
4. Chen, L., Megow, N., Schewior, K.: An o(log m)-competitive algorithm for online machine minimization. SIAM Journal of Computing **47**(6), 2057–2077 (2018)
5. Eberle, F., Megow, N., Schewior, K.: Optimally handling commitment issues in online throughput maximization. In: F. Grandoni, G. Herman, P. Sanders (eds.) European Symposium on Algorithms), *LIPIcs*, vol. 173, pp. 41:1–41:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
6. Im, S., Moseley, B., Pruhs, K., Stein, C.: An o(log log m)-competitive algorithm for online machine minimization. In: 2017 IEEE Real-Time Systems Symposium, pp. 343–350. IEEE Computer Society (2017)
7. Kalyanasundaram, B., Pruhs, K.: Speed is as powerful as clairvoyance. Journal of the ACM **47**(4), 617–643 (2000). Also 1995 Symposium on Foundations of Computer Science
8. Kalyanasundaram, B., Pruhs, K.: Eliminating migration in multi-processor scheduling. Journal of Algorithms **38**(1), 2–24 (2001)
9. Kalyanasundaram, B., Pruhs, K.: Maximizing job completions online. Journal of Algorithms **49**(1), 63–85 (2003). Also 1998 European Symposium on Algorithms
10. Koren, G., Shasha, D.E.: MOCA: A multiprocessor on-line competitive algorithm for real-time system scheduling. Theoretical Computer Science **128**(1&2), 75–97 (1994)
11. Lucier, B., Menache, I., Naor, J., Yaniv, J.: Efficient online scheduling for deadline-sensitive jobs. In: G.E. Blelloch, B. Vöcking (eds.) ACM Symposium on Parallelism in Algorithms and Architectures, pp. 305–314. ACM (2013)
12. Moseley, B., Pruhs, K., Stein, C., Zhou, R.: A competitive algorithm for throughput maximization on identical machines. In: Integer Programming and Combinatorial Optimization - 23rd International Conference, IPCO 2022, *Lecture Notes in Computer Science*, vol. 13265, pp. 402–414. Springer (2022)
13. Phillips, C.A., Stein, C., Torng, E., Wein, J.: Optimal time-critical scheduling via resource augmentation. Algorithmica **32**(2), 163–200 (2002)