Preemptive Online Machine Minimization

Weiqiao Han, Karren Yang*

December 14, 2016

Abstract

In this paper, we will survey recent progress on the preemptive online machine minimization problem. The goal of this machine minimization problem is to determine a feasible schedule for preemptive jobs on a minimum number of parallel machines for jobs with hard deadlines. In the online version of the problem, a job is not visible until its release date, and "preemptive" means a job can be stopped and reassigned to any machine later without losing the processing it has already received. We introduce two basic greedy algorithms, which do not do well on this problem, and analyze two recent algorithms both proposed by Chen et al. [4, 2], which achieve good competitive ratios.

1 Introduction

1.1 Problem Definition

In the machine minimization problem, we are given a set of n jobs, $\{1, \ldots, n\}$. Each job $j \in \{1, \ldots, n\}$ has a **processing time** $p_j \in \mathbb{N}$, a **release date** $r_j \in \mathbb{N}$, which is the earliest possible time at which the job can be processed, and a **deadline** $d_j \in \mathbb{N}$, by which it must be completed. The goal is to use a minimum number of machines such that there is a feasible schedule in which no job misses its deadline. In a **feasible** schedule, each job j is scheduled for p_j units of time within its **time window** $I(j) = [r_j, d_j)$. Each machine can process at most one job at any time, and no job can be simultaneously processed on more than one machine. In the online version of the problem, a job is not visible until its release date. Machine minimization is essential to minimizing resource usage and achieving economic, environmental, and social goals.

In this paper, we introduce algorithms for *preemptive* online machine minimization. Here "preemptive" means that the scheduler is allowed to interrupt the processing of a job (preempt) at any point in time and put a different job on the machine instead. The amount of processing a preempted job already has received is not lost. A preempted job can resume on any machine, and only needs to be processed for its remaining processing time. We analyze the algorithm's performance by competitive analysis. Throughout the paper, we denote by m the number of machines used by an optimal offline solution, or equivalently, we say m is the **optimal offline cost**. Given a set J of jobs, we use $m(J)$ to denote the optimal number of machines to schedule J (in the offline sense).

1.2 Structure of the Paper

In Section 2, we show two basic greedy algorithms for dealing with online machine minimization problems, namely the *Earliest Deadline First* (EDF) algorithm and the *Least Laxity First* (LLF) algorithm. We show that the competitive ratio of EDF is infinite, and the competitive ratio of LLF is worse than a constant [7]. In fact, it has been proven that LLF is $\Omega(n^{1/3})$-competitive and not $f(m)$-competitive for any function computable function f [4, 2], which we will not discuss in detail. In Section 3, we transit from basic algorithms to advanced algorithms. In Section 4, we introduce an $O(\log n)$-competitive algorithm proposed by Chen et al. [4], and in Section 5, we introduce an $O(\log m)$-competitive algorithm [2], invented by the same authors one year later. So far $O(\log m)$ is the best upper bound on the competitive ratio. In Section 6, we briefly summarize the two new algorithms proposed by Chen et al., and hint a few possible extensions.

*{weiqiaoh,karren}@mit.edu

1.3 Related Results

There are some variants of the problem that we are not going to discuss in detail in later sections of this paper. One variant is the machine minimization for *non-preemptive* jobs in which it is necessary to keep a job on a machine, once started, until its completion. In general, this problem is hopeless in terms of competitiveness. In fact, Saha [8] showed that there is no algorithm with competitive ratio $f(m)$ for any computable function f. The case where every job has unit processing time, i.e., $p_j = 1, j = 1, \ldots, n$, has been shown to have an e-competitive algorithm using the Earliest Deadline First (EDF) algorithm [1, 6]. However, for jobs with unit processing time, there is no difference between preemptive jobs and non-preemptive jobs.

Another variant of the problem is the machine minimization for *preemptive non-migratory* jobs, where "non-migratory" means each job is processed by exactly one machine, i.e., a preempted job can only resume on the machine it started with. In practice, while migratory schedules may be easier to design and have a better performance, it may also cause a significant overhead in communication and synchronization, and increase the risk of cache-failures, so non-migratory schedules are preferred. Assuming $p_j, r_j, d_j \in \mathbb{Q}$ instead of $\mathbb{N}$, Chen et al. [3] showed that there is an example of n jobs, which can be processed by a non-migratory offline schedule on m machines, but any (deterministic) non-migratory online schedule requires at least $\Omega(\log n)$ machines. This result tells us about the power of migration, which may help explain the superior performance of the $O(\log m)$-competitive algorithm over the $O(\log n)$-competitive algorithm.

2 Basic Greedy Algorithms

2.1 Earliest Deadline First (EDF)

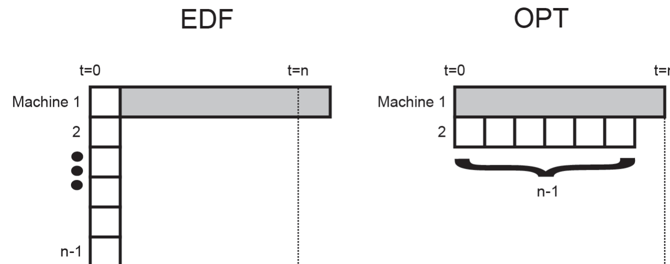
We start with the Earliest Deadline First (EDF) algorithm.

Algorithm 2.1 (EDF [6]). *Suppose we have $m(t)$ available machines at time t. Then EDF schedules the $m(t)$ available jobs with the earliest deadlines, where "available jobs" means released but not completed jobs.*

EDF works well for certain instances, for example when jobs have unit processing time, i.e., $p_j = 1, \forall j$, because the penalty for doing the wrong job is very low. However, EDF easily fails on jobs with arbitrary processing time, as shown by the following theorem.

Theorem 2.1 ([7]). *For any n, there exists a set of n jobs that causes EDF to fail on $< n$ machines, given that $m \geq 2$.*

Proof by picture. Consider the following set of n jobs all released at time 0: 1 large job due at time n with a processing time of n; and $n-1$ small jobs due at time $n-1$ with processing time of 1. EDF does not provide a feasible schedule on fewer than n machines because it prioritizes the small jobs, but a feasible schedule exists for $m = 2$ machines.



□

Intuitively, EDF performs poorly on this input because it fails to discern that the large can only be delayed by a short amount of time. On the other hand, each of the small jobs can be can be delayed for quite a long time. We formalize this useful concept below.

Definition 2.1. *For each job j, we define its* ***laxity*** *$l_j = d_j - r_j - p_j$ as the maximum amount of time that the job can be put off without missing its deadline.*

We can view the laxity of a job as the maximum "budget" for delaying a job. Whenever a job is not processing during its time window, its budget is charged by the amount of this delay.

2.2 Least Laxity First (LLF)

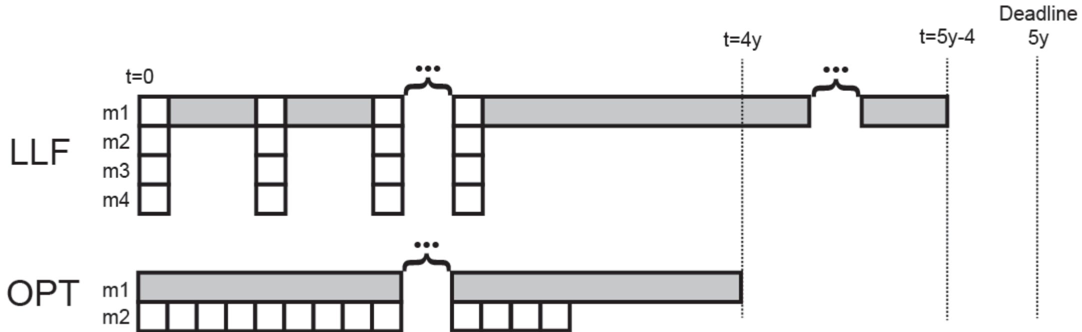
Now that we have the distinction between jobs with more laxity and jobs with less laxity, a logical improvement on EDF would be to schedule the jobs with less laxity before the ones with more laxity. This is the basis of the Least Laxity First (LLF) algorithm, which was the best known algorithm for this scheduling problem until recently [7].

Algorithm 2.2 (LLF [2])**.** *Suppose we have $m(t)$ available machines at time t. Then LLF schedules the $m(t)$ available jobs with the smallest laxities.*

While better than EDF, LLF is still vulnerable to certain inputs.

Theorem 2.2 ([7])**.** *LLF is $\omega(1)$-competitive; for any integer c, there is a sequence of inputs for which LLF fails to find a feasible schedule on cm machines.*

Proof by an example. We give a loose sketch of the proof when $c = 2, m = 2$ and LLF fails to find a feasible schedule on 4 machines. Let y be a large constant. Consider the following job set: one $4y$-length job with release time 0, deadline $5y$ and therefore laxity y; and $y - 4$ sets of 4 small jobs with processing times of 1, time windows of length 4, and laxity 3, which are released at times 0, 4, 8, ... $4y - 16$. Since y is a large constant, the smaller jobs are always prioritized by LLF and delay the large job. Therefore, the optimum schedule completes the jobs by time $4y$, but LLF needs $y - 4$ of extra time as shown in the picture below.



Since LLF still has to process its large job in time $[4y, 5y]$ while the optimal schedule is free, at time $4y$, we present both algorithms with a scaled-down version of this same job set (i.e. with $y' = y/5$) that falls in the interval $[4y, 5y]$. Again, the optimal algorithm will finish before LLF does. We recursively add scaled-down job sets in this manner until LLF is overloaded and cannot complete the all of the large jobs in each scaled copy before the deadline $5y$. □

LLF performs poorly in this example because it only considers absolute remaining laxity, ignoring the amount of time that jobs have already spent in the system. As a result, large jobs with large laxity can be delayed until it is too late, in favor of processing batches of smaller jobs with smaller laxity.

3 Advanced Algorithms

Now we are sufficiently motivated to discuss the two recent algorithms of Chen et al. [4, 2]. These algorithms overcome the limitation of LLF by considering laxity in relative instead of absolute terms. Specifically, they attempt to keep the remaining laxity of each job above a certain fraction of its initial level. The first algorithm has a competitive ratio of $O(\log n)$, which depends on the total number of jobs released. The second algorithm achieves a competitive ratio of $O(\log m)$, which only depends on the optimal offline cost, and is currently the best known algorithm for online machine minimization.

Both algorithms divide jobs into "loose" jobs (jobs with a small processing time relative to its entire time window) and "tight" jobs and process them separately. Notice that these concepts depend on relative laxity instead of absolute laxity. We make this notion clear with the following definitions:

Definition 3.1. *A job j is **α-loose** if, for some $\alpha \in (0, 1)$, we have $p_j \le \alpha(d_j - r_j)$ (alternatively, $l_j \ge (1 - \alpha)(d_j - r_j)$), and is **$\alpha$-tight** otherwise. We also call α-loose jobs the **flexible** jobs and α-tight jobs the **critical** jobs.*

The α-loose jobs are easily processed using EDF because of the following theorem:

Theorem 3.1 ([4]). *If every job is α-loose, then EDF is $\frac{1}{(1-\alpha)^2}$-competitive.*

We do not prove this theorem, but note that it makes sense intuitively because the penalty for processing the wrong job is bounded when all jobs are loose.

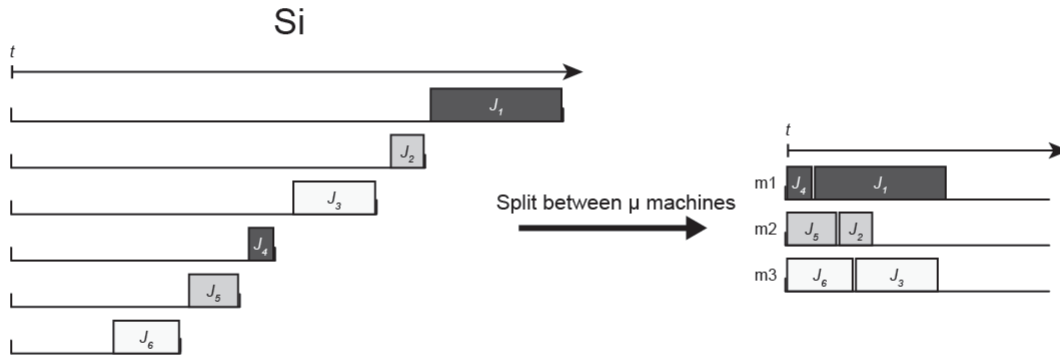
Separately, the α-tight jobs are processed using more advanced schemes, which we describe in detail below.

4 An $O(\log n)$-Competitive Algorithm

This algorithm is a clever extension of the EDF algorithm. The basic idea is that we divide the α-tight jobs into several groups so that there is a comfortable time buffer between adjacent jobs in the same group, with more critical jobs put in smaller groups. Each group is scheduled on a different machine using EDF. Because of the buffer, earlier jobs do not delay later jobs significantly relative to the amount of laxity that the later jobs have, so no job spends so much of its laxity that it becomes problematic when more jobs are released. This allows the algorithm to achieve a competitive ratio of $O(\log n)$, which is superior to the standard EDF and LLF algorithms.

4.1 Description of Algorithm

Algorithm 4.1. *Choose some $0 < \alpha < 1/2$. Since jobs that are α-loose can be processed by EDF on separate machines, we focus on processing only the α-tight (i.e. critical) jobs. Let X_t denote the pool of critical jobs at time point t, and suppose that t is the release date of some α-tight job. At the start of a time interval $[t, t+1)$, we obtain X_t from X_{t-1} by adding any new α-tight jobs and removing jobs that are no longer available or that have become α-loose. Those jobs that are α-loose are transferred to the other pool, which we call the "safe" pool, to be processed by EDF. We divide X_t into groups as follows. First, we sort the jobs in X_t by their deadlines. Then starting from the job with the latest deadline, we assign each job j to an arbitrary group S provided that $d_j - t \leq \min_{k\in S} l_k(t)$, where $l_k(t)$ denotes the remaining laxity of job k at time t, i.e., we only assign j to S if the other jobs in S have enough laxity to be delayed until after j's deadline. If no such S exists, then we create a new set for this job. Repeating those process for all the jobs, we end up with h_t groups, $S_1, \ldots, S_{h_t}$, such that the jobs in each S_i can be temporally distributed as shown to the left part of the figure below. Note that each S_i can feasible be scheduled on one machine.*



This next step is key. For each S_i, instead of scheduling the jobs on one machine, we rank the jobs $j_1 \ldots j_{|S_i|}$ by their deadlines from latest to earliest and distribute them between $\mu_t = O(\log|J(t)|)$ machines as shown in the right part of the figure above, where $J(t)$ is the set of jobs released by time t. Specifically, each j_i is assigned to machine $((i-1) \mod \mu_t + 1)$. This step is important because it ensures that later jobs do not lose too much of their initial laxity waiting for earlier jobs to finish, since there is always some laxity buffer between adjacent jobs. We then schedule and process the jobs on each machine using EDF. If a job becomes α-loose, we remove it from its machine, transfer it to the safe pool, and continue to process the critical jobs

on their current machines. If a new α-tight job is released, we recompile the pool of critical jobs and restart the process of assigning jobs to machines.

4.2 Analysis of Algorithm

Theorem 4.1 ([4]). *The algorithm described above is $O(\log n)$-competitive.*

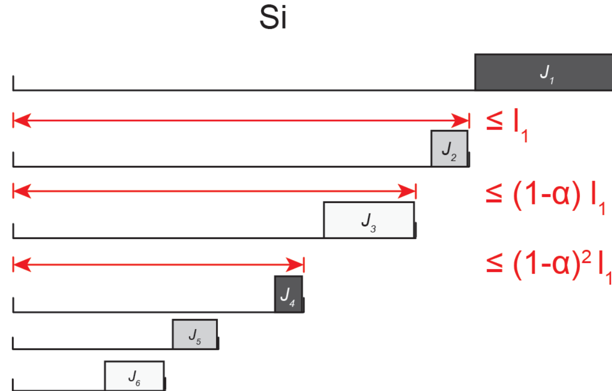
Let L be the set of α-loose jobs that enter the safe pool, and let $m(L)$ be the optimal number of machines for processing L. By Theorem 3.1, we will use $O(m(L))$ machines to process the safe jobs. To process the critical pool, we use $h_t\mu_t$ machines between adding new jobs, so the number of machines it needs is $\max_t(h_t\mu_t)$. So the total number of machines we need is $O(m(L) + \max_t(h_t\mu_t))$.

The outline of the proof is as follows. If we choose μ_t just large enough $(= O(\log|J(t)|) = O(\log n))$ every time we construct new groups of jobs, then the jobs in each S_i can be assigned to μ_t machines so sparsely that the laxity of each job at any time t never drops below a constant fraction β of its initial value (Lemma 4.2) Decreasing all jobs' laxities by at most a factor of β increases the optimal offline cost by at most a constant factor $O(1/\beta)$ (Lemma 4.3). Finally, we use this to upper bound $m(L)$ and h_t by $O(m/\beta) = O(m)$, which gives the desired result that the algorithm succeeds on $O(m \log n)$ machines.

Lemma 4.2 ([4]). *For some $\mu_t = O(\log|J(t)|)$, we have $l_j(t) \geq \beta l_j$ for some $\beta \in (0, 1)$, for all times t and jobs $j \in X_t$.*

Proof. In order for the laxity of some job j to be reduced, j must be delayed in favor of processing another job. Since the jobs assigned to each machine are scheduled by EDF, job j is delayed for the time it takes to run jobs with earlier deadlines that have been assigned to the same machine.

Let $S_i = \{j_1, j_2, ..., j_{|S_i|}\}$, where the j_i are ranked by their deadlines from the latest (j_1) to the earliest $(j_{|S_i|})$. If j_k directly precedes j_i in the same machine, then $k = i + \mu_t$, and we can bound the time window length of j_k by $d_{j_k} - t \leq (1-\alpha)^{\mu_t - 1} \cdot l_{j_i}(t)$, which is obtained by repeatedly applying the two inequalities $d_{j_{i+1}} - t \leq l_{j_i}(t)$ (by construction of S_i) and $l_{j_i}(t) \leq (1-\alpha)(d_{j_i} - t) \leq (1-\alpha)l_{j_{i-1}}(t)$ (since jobs are α-tight). This is illustrated in the figure below, with $j_i = J_1$, $j_k = J_4$ and μ_t=3.



Choosing $\mu_t = c\log|J(t)| + 1$, for $c > 2\log(1/(1-\alpha))$, gives us $(1-\alpha)^{\mu_t - 1} \leq |J(t)|^{-2}$, where $J(t)$ is the set of all jobs released by time t. Substituting this into the inequality yields an upper bound on the window length of job j_k, which upper bounds the delay of job j_i:

$$delay(j_i) \leq d_{j_k} - t \leq (1-\alpha)^{\mu_t - 1} \cdot l_{j_i}(t) \leq (1-\alpha)^{\mu_t - 1} \cdot l_{j_i} \leq \frac{l_{j_i}}{|J(t)|^2}.$$

Since we recompile whenever new jobs are released, to bound the total delay of j_i, we sum the delays over all possible values of $J(t)$ from 2 to n. (We exclude $|J(t)| = 1$ since if there is only 1 job, then there is no second job to delay it.) This gives us:

$$total.delay(j_i) \leq \sum_{|J(t)|=2}^{n} \frac{l_{j_i}}{|J(t)|^2} < \sum_{|J(t)|=2}^{\infty} \frac{l_{j_i}}{|J(t)|^2} < 0.65 \cdot l_{j_i}$$

j_i is delayed by at most 0.65 of its initial laxity over the entire schedule, which means it always keeps at least 0.35 of its initial laxity. Since j_i was chosen arbitrarily, this concludes the proof with $\beta = 0.35$.

□

Now, we claim that decreasing all jobs' laxities by at most a factor of β increases the optimal number of machines by at most a constant $O(1/\beta)$. This is the most important and challenging proof of the paper, so in places, we have greatly simplified the argument.

Lemma 4.3 ([4])**.** *Suppose we decrease the laxity of every job $j \in J$ by a factor of β by "trimming" it from the left. Specifically, let trimmed job $j^\beta \in J^\beta$ have processing time p_j, release time $r_j + (1-\beta)l_j$, and deadline d_j. Then the inequality $m(J^\beta) \leq m(J)/\beta$ holds, where $m(J^\beta)$ and $m(J)$ are the optimal number of machines for scheduling the given job sets J^β and J respectively.*

Before proving the lemma, we first provide a new characterization of the optimal number of machines $m(J)$ for any job set J. Suppose that for any time interval I, we are as "lazy" as possible, i.e. we use as much laxity as each job has to offer to push its workload outside of the interval I. This gives us the following very intuitive definition:

Definition 4.1. *Let j be a job and let $I(j) = [r_j, d_j)$ be the time window of j. Let $|I|$ indicate the length of an interval I (which does not have to be continuous). Then the **minimum workload contribution** of j to a time interval I is defined as $C(j, I) := \max\{0, |I \cap I(j)| - l_j\}$. In other terms, $C(j, I)$ is the minimum processing time that j must receive in the interval I, having used as much laxity as possible in this interval.*

For any time interval I, even if we were to be as "lazy" as possible, we would still have to do $\sum_{j\in J} C(j, I)$ amount of processing to avoid missing any deadlines. So $\sum_{j\in J} C(j, I)/|I|$ gives a lower bound on $m(J)$, for any interval I. The maximum over all such intervals I is still a lower bound on $m(J)$. The following theorem, which we state without proof, says that this lower bound is tight:

Theorem 4.4 ([4])**.** *Let J be a set of jobs and m be the optimum number of machines. Then we have*

$$m(J) = \left\lceil \max_I \sum_{j\in J} \frac{C(j, I)}{|I|} \right\rceil$$

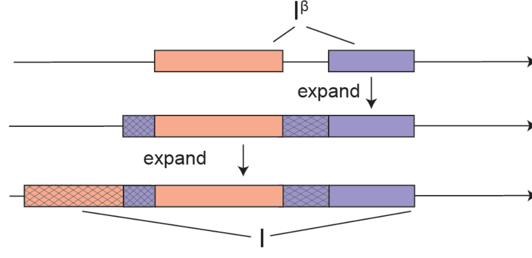
In other terms, if we were to consider every time interval and defer as much workload outside of this interval as we can, then m would be the number of machines required for the worst time interval.

Proof of Lemma 4.3. Based on Theorem 4.4, there is some interval I^β such that

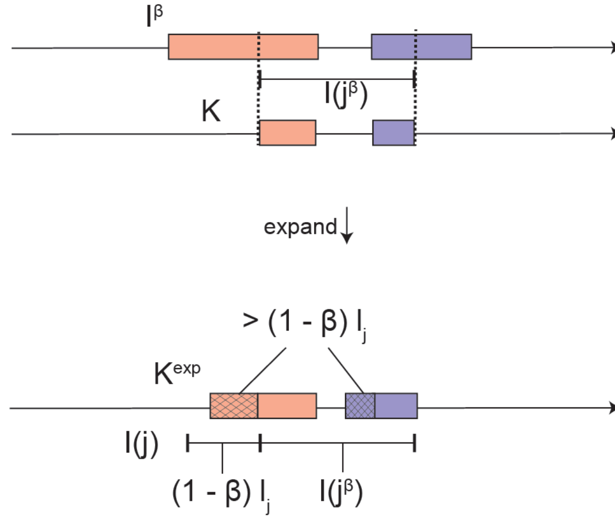
$$m(J^\beta) = \left\lceil \sum_{j^\beta} \frac{C(j^\beta, I^\beta)}{|I^\beta|} \right\rceil.$$

i.e., I^β is the worst interval for the trimmed job set J^β. We are going to find some interval I such that $|I^\beta| = \beta|I|$ and $C(j, I) \geq C(j^\beta, I^\beta)$ for every job. This means that I is $1/\beta$ longer than I^β but receives more work from job set J than I^β receives from job set J^β. Therefore, J^β must require no more than $1/\beta$ times more machines than J. Algebraically, this is written as: $m(J^\beta) = \left\lceil \sum_{j^\beta} \frac{C(j^\beta, I^\beta)}{|I^\beta|} \right\rceil \leq \left\lceil \frac{1}{\beta} \sum_j \frac{C(j, I)}{|I|} \right\rceil \leq m(J)/\beta$.

We show that we can get such an interval I by expanding I^β to the left as follows (see figure below for demonstration). We first represent I^β as disjoint continuous intervals $[a_1, b_1), [a_2, b_2), ...[a_k, b_k)$. Starting from the rightmost interval, we try to expand it by a factor of $1/\beta$ by moving the left boundary to the left, but stop if we encounter the next interval over: expanded$[a_k, b_k) = [\max\{b_k - (b_k - a_k)/\beta, b_{k-1}\}, b_k)$. If we are unable to expand this interval by $1/\beta$, then we expand the next interval to the left by a factor of $1/\beta$ plus the difference $b_{k-1} - (b_k - a_k)/\beta$. We iterate until we have expanded all the intervals by a total factor of $1/\beta$. Let I be the union of the expanded intervals, and note that we have $|I^\beta|/\beta = |I|$ as desired.



Next, we show that $C(j, I) \geq C(j^\beta, I^\beta)$ for every job. Recall $C(j^\beta, I^\beta) = \max\{0, |I^\beta \cap I(j^\beta)| - l_{j^\beta}\}$, i.e., $C(j^\beta, I^\beta)$ is the minimum workload contribution of job j^β to the time interval I^β. For each j^β, we need to focus on the portion of I^β that is relevant to it, so let $K = I^\beta \cap I(j^\beta)$ (see figure below). If $|K| \leq l_{j^\beta}$, then j^β does not have to be processed in K and $C(j^\beta, I^\beta) = 0$, which makes the proof that $C(j, I) \geq C(j^\beta, I^\beta)$ trivial. Therefore, we focus on the case where $|K| > l_{j^\beta}$, i.e., some of j^β must be processed in K.



Suppose we expand the set K to the left by a factor of $1/\beta$ in the same way that I^β was expanded before, and represent this expanded set as K^{exp} (see figure above for illustration of this section). Because $|K| > l_{j^\beta} = \beta l_j$, then we have $|K^{exp}| - |K| = (\frac{1}{\beta} - 1)|K| > (1 - \beta)l_j$, meaning that K^{exp} is K expanded to the left by more than $(1 - \beta)l_j$. Recall that by definition $I(j)$ is $I(j^\beta)$ expanded to the left by $(1 - \beta)l_j$. We claim that the expanded portion of K overlaps with $I(j)$ by at least $(1 - \beta)l_j$ (see the figure above). Therefore, we have $|K^{exp} \cap I(j)| \geq |K| + (1 - \beta)l_j$.

Since $K^{exp} \subseteq I$, we get the result

$$C(j, I) \geq C(j, K^{exp}) = |K^{exp} \cap I(j)| - l_j \geq |K| - \beta l_j = C(j^\beta, I^\beta)$$

which concludes the proof. □

Next, we claim that the number of groups $S_1, \ldots, S_{h_t}$ that are formed using our algorithm is not excessive.

Definition 4.2. *The* ***residue*** *of a job j at time t is a job with time interval $[t, d_j)$ and processing time $p_j(t)$, the remaining processing time of job j at time t.*

Lemma 4.5. *At all times that we create new sets $S_1, \ldots, S_{h_t}$ during the execution of the algorithm, we have $h_t = O(m(Y_t))$, where Y_t is the set of the residues of the critical jobs X_t at time t.*

We omit the simple proof of this lemma in favor of an intuitive explanation. Recall that when we form the groups S_i from the job residues Y_t, we try to pack as many jobs into each group as possible, with the requirement that the jobs in each group can feasibly be scheduled on one machine by EDF (recall the figure

of S_i from the description of the algorithm). Since the jobs are α-tight, we can convince ourselves that this packing does not waste too much space. Therefore, the number of groups S_i that our algorithm forms is not too far from the number of machines required for the optimal packing of Y_t, which gives us $h_t = O(m(Y_t))$.

We are now ready to prove the main result.

Proof of Theorem 4.1. Recall from the beginning of this subsection that we need $O(m(L) + \max_t(h_t \cdot \mu_t))$ machines for the algorithm to work. We prove that this is $O(m \log n)$ if we choose $\mu_t > 2 \log \frac{1}{1-\alpha} \log |J(t)| = O(\log n)$.

According to Lemma 4.5, h_t is bound above by $O(m(Y_t))$. We now show that $m(L)$ and $m(Y_t)$ are both $O(m)$. Recall that L is the set of jobs that enter the "safe" pool, consisting of those jobs that were either α-loose upon release or later transferred from the "critical" pool. Also, recall that Y_t is the residues of the jobs in the "critical" pool at time t. Therefore, both L and Y_t contain a subset of the original jobs J, with the modification that some jobs may have been delayed or partially-processed in the "critical" pool, i.e., the time windows of some of these jobs are "trimmed" from the left.

From Lemma 4.2, we know that based on our choice of μ_t, the laxities of the jobs in the critical pool never fall below some constant fraction $\beta \in (0, 1)$ of their initial values. It follows that L and Y_t are no harder to schedule than the original jobs if they were trimmed from the left by $1 - \beta$ of their laxity. Then according to Lemma 4.3, both $m(L)$ and $m(Y_t)$ are $\leq m(J^\beta) = O(m/\beta) = O(m)$. This completes the proof that the algorithm is $O(\log n)$-competitive. □

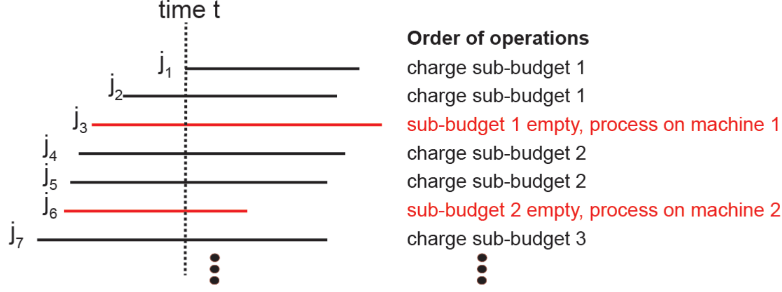
5 An $O(\log m)$-Competitive Algorithm

We are finally ready to discuss the best known algorithm, which builds on the basic idea of LLF. Similar to the $O(\log n)$-competitive algorithm, this algorithm performs well because it attempts to keep the remaining laxity of each job above a certain fraction of its initial level. Recall from Section 2.2 that LLF prioritizes jobs with the least laxity, but sometimes jobs with large laxities can be delayed for too long. This causes problems if urgent jobs keep arising and prevent these jobs from being completed on time. To overcome this problem, Chen et al. [2] propose a method to intelligently distribute the total laxity of a job. When a job is released, its laxity "budget" is divided into $m' + 1$ "sub-budgets", where m' is the number of machines we choose to use. A job that is waiting to be processed on a machine is prioritized once it depletes its sub-budget for that machine, instead of having to deplete its entire budget. This way, no job is delayed for too long or depletes too much of its initial laxity, which allows the algorithm to achieve a competitive ratio of $O(\log m)$.

5.1 Description of Algorithm

Algorithm 5.1 ([2])**.** *Choose some $0 < \alpha < 1/2$. Since jobs that are α-loose can be processed by EDF on separate machines with $O(1)$-competitive ratio (Theorem 3.1), we focus on processing only the α-tight (i.e. critical) jobs. At the start of the algorithm, we open some $m' = O(m \log m)$ machines, numbered 1 through m'. The choice of m' will become clear in the later analysis. Whenever a new α-tight job j is released, we divide its laxity into $m' + 1$ equal "sub-budgets" of $l_j/(m' + 1)$ and assign each sub-budget to a machine.*

Suppose we are to assign jobs to machines at time t. Let X_t be the available critical jobs at time t. WLOG, $X_t = \{j_1, j_2, ..., j_{|X_t|}\}$ where $r_{j_i} \geq r_{j_{i+1}}$, i.e., the j_i are sorted by release date from the latest (j_1) to the earliest ($j_{|X_t|}$). To assign some job to machine 1, we start at the beginning of X_t and check the 1st sub-budgets of the jobs in order. As long as the 1st sub-budget of the job is positive, we delay that job and check the next job. Whenever we delay a job, we charge that delay to its first sub-budget. Once we find a job $j_{a(1)}$ whose 1st sub-budget is 0, then we assign $j_{a(1)}$ to machine 1. Next, we find a job to assign to machine 2. We consider jobs that come after $j_{a(1)}$ in X_t and now we check the 2nd sub-budgets of the jobs in order. Once we find a job $j_{a(2)}$ whose 2nd sub-budget is 0, then we assign $j_{a(2)}$ to machine 2 and charge the 2nd sub-budgets of the jobs that we choose to delay. We continue assigning jobs to machines in this manner until we have made a full pass through X_t (see figure below for demonstration).



Once we have finished assigning jobs to machines, we "start the clock" on processing. All jobs that are not assigned to machines have this time interval charged to some sub-budget, i.e. some sub-budget gets depleted as processing time passes for the assigned jobs. Once the sub-budget of some job becomes 0 or there is an event such as a job release or a deadline at time t', then we "stop the clock" and repeat the assignment process for $X_{t'}$.

5.2 Analysis of Algorithm

Since this algorithm is $O(\log m)$-competitive and opens all of its machines at the beginning, it requires prior knowledge of the actual minimum number of machines m needed to process the input jobs (i.e. it is a semi-online algorithm). We usually cannot get this information until after all the jobs have been released. Fortunately, there is a doubling scheme which allows us to use any semi-online algorithm to solve the online problem:

Theorem 5.1 (doubling scheme [5]). *Given a $f(m)$-comptitive algorithm for semi-online machine minimization, where $f(\cdot)$ is a non-decreasing function, there is a doubling-based algorithm that is $4f(2m)$-competitive for online machine minimization. (The only difference between the semi-online algorithm and the online algorithm is that the semi-online algorithm knows the optimal minimum number of machines m in advance.)*

Note: In [2], the doubling scheme is stated as "THEOREM 2.2: Given a ρ-competitive algorithm for semi-online machine minimization, there is a doubling-based algorithm that is 4ρ-competitive for online machine minimization", the proof of which is given in [4], and the authors conclude that they "may assume that the optimum number of machines m is known in advance by losing at most a factor 4 in competitive ratio". However, the proof in [4] doesn't work in this case, because the competitive ratio is a function of m. Here, we give a proof for a slightly difference result that is applicable to semi-online algorithms with non-constant competitive ratios.

Proof. Let $A(m)$ denote the $f(m)$-competitive algorithm for the semi-online machine minimization given the optimum number of machines m. Let $m(t)$ denote the optimum number of machines for the set $J(t)$ of all jobs released so far. So $m(t)$ can be computed by the online algorithm at time t. Then our online algorithm is

- Let $t_0 = \min_{j\in J} r_j$. For $i = 1, 2, \ldots$, let $t_i = \min\{t | m(t) \geq 2m(t_{i-1})\}$, i.e., t_i's are the time points when the optimal cost $m(t)$ doubles.
- At any time $t_i, i = 0, 1\ldots$, open $2m(t_i)f(2m(t_i))$ additional machines, and schedule all jobs with $r_j \in [t_i, t_{i+1})$ by the Algorithm $A(2m(t_i))$.

Since during the time interval $[t_i, t_{i+1})$, the optimum number of machines $m(t) \leq m(t_{i+1}) = 2m(t_i)$, $A(2m(t_i))$ is sufficient to schedule all jobs with $r_j \in [t_i, t_{i+1})$ using the $2m(t_i)f(2m(t_i))$ additional machines. So the total number of machines is bounded by

$$\sum_{t_i} 2m(t_i)f(2m(t_i)) = \sum_{k:2^k\leq m} 2\cdot 2^k f(2\cdot 2^k) \leq 2(\sum_{k:2^k\leq m} 2^k)f(2m) = 2(2^{1+\lfloor \log_2 m\rfloor} - 1)f(2m) \leq 4mf(2m).$$

Therefore, the competitive ratio of the online algorithm is $4f(2m)$. $\square$

According to Theorem 5.1, if we can show that a semi-online algorithm is $O(\log m)$-competitive, then there is an online algorithm using the doubling scheme that is still $O(\log m)$-competitive. Therefore, we just need to prove the following theorem:

Theorem 5.2 ([2])**.** *The (semi-online) algorithm 5.1 described above is $O(\log m)$-competitive.*

Recall that we only deplete from a sub-budget if it is positive. Therefore, no job's budget becomes negative over the course of the algorithm. The only way that the algorithm could fail is if some job's $(m'+1)$-th sub-budget reaches 0. If this does happen, then the algorithm fails because there is no machine to assign that job to. To prove that this does not happen, we show that the maximum number of "intersecting" jobs (i.e., jobs whose sub-budget is 0 and need to be assigned to a machine) is upper bound by $O(m \log m')$ if the algorithm fails, so choosing a large enough value of m' such that $m' \geq O(m \log m')$ ensures that the algorithm never fails.

First, we give a new lower bound on the optimal offline cost m that depends on the number and laxity of jobs covering a given time interval. Note that a job j is said to **cover** a time point t if $t \in I(j) = [r_j, d_j)$.

Definition 5.1 ([2])**.** *Let G be a set of α-tight jobs and let T be a time interval (not necessarily continuous). We say that (G, T) is* **(μ, β)-critical** *if for some $\mu \in \mathbb{N}$ and some $\beta \in (0, 1)$,*

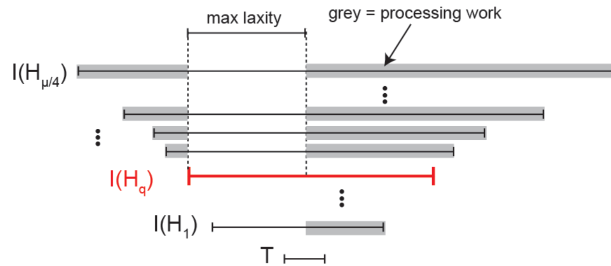
(i) each time point $t \in T$ is covered by $\geq \mu$ jobs
(ii) $|T \cap I(j)| \geq \beta l_j$ for any $j \in G$

Theorem 5.3 ([2])**.** *Suppose (G, T) is a (μ, β)-critical pair. Then*

$$m = \Omega\left(\frac{\mu}{\log 1/\beta}\right)$$

First, we explain intuitively why this bound makes sense. As μ increases, the number of jobs that cover the same time interval T increases, so the optimal number of machines should increase as well. When β is close to 0, this means that the $|T|$ could be much smaller than the total laxity of the jobs, so the jobs might have a lot of flexibility and the lower bound on the optimum number of machines is small. On the other hand, when β is close to 1, this means that $|T|$ is at least on the same order as the total laxity of the jobs, so the jobs have less flexibility; as a result, the lower bound on the optimal number of machines is high. Clearly, to get the tightest lower bound as possible, we would want to pick the largest μ and β that satisfy the conditions.

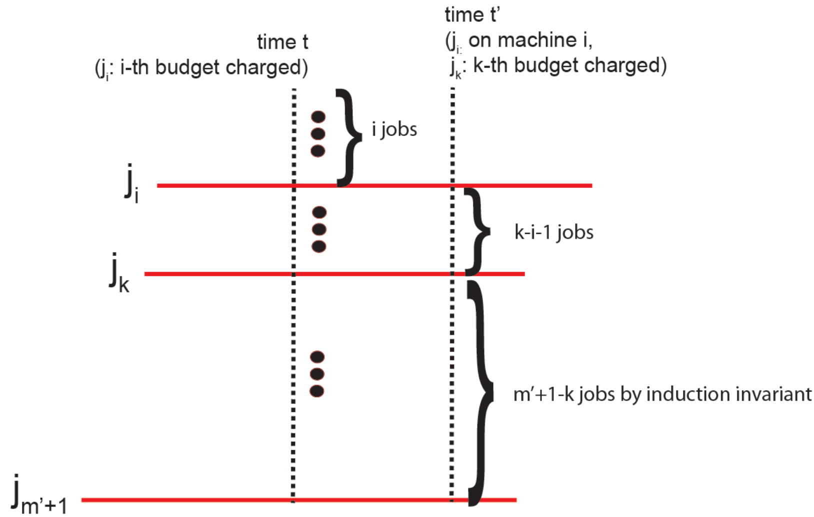
The gist of the proof is as follows (see figure below). From the jobs in G, we can construct a sets of pairwise-disjoint job intervals that overlap T. Condition (ii) gives us a sense of how much laxity these intervals have on average, so we can pick a subset (specifically, $\mu/4$) of these intervals (denoted $I(H_i)$) with bounded laxity. Taking advantage of the bounded laxity of the intervals, we find one interval $I(H_q)$ whose index is $q = O(m \log 1/\beta)$, with the property that every larger interval has a significant workload contribution to $I(H_q)$. If q is much smaller than $\mu/4$, then the interval $I(H_q)$ will be overloaded with work, and it is not possible to complete it with m machines. Therefore, q must be relatively close to $\mu/4$. We can think of this as $q = O(m \log 1/\beta) = \Omega(\mu/4)$, which gives us the theorem after some minor rearrangement.



Corollary 5.3.1 ([2])**.** *Suppose (G, T) is a (μ, β)-critical pair. Then $\mu = O(m \log(1/\beta))$ results directly from Theorem 5.3.*

Proof of Theorem 5.2. Suppose that the set of jobs J fails on the above algorithm. Then we want to find a subset of jobs and a set of time points (F, T) that are responsible for the failure of j. We can look at the history of the schedule to determine which jobs are responsible. When the algorithm fails, it is because the $(m'+1)$-th sub-budget of some job $j^* \in J$ reaches 0. The $(m'+1)$-th sub-budget was charged whenever jobs with lower indices than j^* were occupying the m'-th machine. It follows that those jobs and the time they are on the m'-th machine are responsible for the failure. Similarly, those jobs were being processed on the m'-th machines only because their m'-th sub-budgets reached 0. Therefore, the jobs with lower indices occupying the $(m'-1)$-th machine at the times that their m'-th sub-budgets were charged are also responsible for the failure. Using this reasoning recursively, we arrive at a failure set (F, T), where F includes all the jobs that are responsible in the failure and T consists of all the times that the responsible sub-budgets of the jobs in F were being charged.

We complete the proof by showing that (F, T) is always a $(m'+1, 1/(m'+1))$-critical pair. Condition (ii) of the definition is obviously satisfied, since each $j \in F$ runs out of one sub-budget during T, implying that $|T \cap I(j)| \geq 1/(m'+1) \cdot l_j$. Condition (i) is satisfied since each $t \in T$ is covered by at least $m'+1$ distinct jobs in F, and the proof of this is as follows. To make it more clear, we have also provided an illustration below.



By definition, $t \in T$ implies that there is some job j_i in F whose i-th sub-budget is charged at time t. Since the i-th sub-budget is charged only if $i-1$ machines are being occupied by jobs with lower indices than j_i, then including j_i we have i jobs that cover t. We now need to show that at least $m'+1-i$ jobs with higher indices than j_i also cover t. Recall that the jobs are indexed by their release dates from the latest to the earliest, so jobs with higher indices than j_i are released before j_i. Therefore, we just need to show that $m'+1-i$ jobs with higher indices than j_i are still available at some time $\geq t$. To do this, we trace a path of jobs from j_i to the job that ultimately causes the failure, $j_{m'+1}$ and use backwards induction. For $j_{m'+1}$, the invariant is satisfied since at least 0 jobs in F with higher indices than $j_{m'+1}$ are available at time $\geq t$. Now suppose that the invariant holds for all j_k on the paths between j_i and $j_{m'+1}$, and we will prove that it holds for j_i. By definition of $j_i \in F$, at some time $t' > t$, job j_i is being processed on the i-th machine, which causes the k-th sub-budget of a higher index job, $j_k \in F$, to be charged. Since the k-th sub-budget is only charged if $k-1$ machines are being occupied by jobs with lower indices than j_k, and exactly i of those are occupied by jobs with index j_i or lower, then there must be $k-i-1$ distinct jobs occupying machines with indices between j_i and j_k at time t'. According to the induction invariant, there are $m'+1-k$ jobs with indices higher than j_k that are available at time $\geq t$. Therefore, there are total of $m'+1-i$ jobs in F with higher indices than j_i available at some time $\geq t$, which completes the proof that condition (i) is satisfied.

Corollary 5.3.1 implies that for an input to fail on our algorithm, we must have $m' \leq Cm\log(m')$ for a specific constant C. But if we choose $m' = C'm\log m$ with $C' \gg C$, then this inequality no longer holds and the algorithm cannot have a failure set. Therefore, for $m' = O(m\log m)$, our algorithm feasibly schedules all of the jobs. $\square$

6 Discussion and Conclusion

In this paper, we analyzed two greedy algorithms (EDF, LLF) and two recent advanced algorithms for online machine minimization for preemptive jobs [4, 2]. EDF performs poorly because it only schedules based on deadlines, so it does not know to prioritize jobs with more laxity over jobs with less laxity. LLF improves on EDF by scheduling the jobs with least remaining laxity. Unfortunately, the downside of LLF is that some jobs can be procrastinated for too long. The recent algorithms by Chen et al. [4, 2] perform better than EDF and LLF by considering laxity in relative rather than absolute amounts; these algorithms try to keep the remaining laxity of jobs from dropping below a certain fraction of their initial levels. The $O(\log n)$-competitive algorithm accomplishes this by distributing the jobs across just the right number of machines, so that jobs with earlier deadlines only takes up a fraction of the laxity of later jobs. The $O(\log m)$-competitive algorithm accomplishes this by dividing the laxity budget of each job into sub-budgets for each machine, and prioritizing jobs every time they use up one of their sub-budgets. Therefore, each job only uses up a fraction of its total laxity before it is prioritized for processing.

After analyzing these algorithms, we are left with three main questions. First, we are curious about the main reason for the difference in performance between the $O(\log n)$ and $O(\log m)$ competitive algorithms, given that they share the same general recipe for success in terms of managing laxity. We surmise that the reason may be due to the forced migration of jobs under the $O(\log m)$ competitive algorithm. Since jobs are indexed by their release dates, as jobs get older, they are forced to migrate towards higher-numbered machines, while lower-numbered machines usually service the most recent jobs. Under the $O(\log n)$ competitive algorithm, however, jobs are indexed by deadline. It is possible for an adversary to provide a job set with earlier and earlier deadlines, such that the jobs do not migrate much between machines. We hypothesize that the forced migration of jobs by the $O(\log m)$-competitive algorithm makes it better at distributing work than the $O(\log n)$-competitive algorithm, particularly when faced with challenging inputs.

The second question that we have is, in which situations is it actually worth it to apply these recent algorithms. EDF and LLF are simple greedy algorithms that do not cost much resources to perform scheduling. However, the $O(\log n)$ and $O(\log m)$-competitive algorithms are much more computationally expensive. For example, the $O(\log n)$-competitive algorithm frequently checks for α-loose jobs to remove from the critical pool of jobs. The $O(\log m)$-competitive algorithm is a semi-online algorithm that is made to be an online algorithm using the doubling scheme. Whenever a job is released, the offline optimum of the jobs so far must be computed, which can get expensive over time.

The final question that we are left with after reading these papers is also the central question of 6.854, which is – can we do better? It remains an open question whether there is a constant competitive online algorithm. The authors of the two advanced algorithms believe there to be a deterministic one, although they cannot show this. Our hunch is that the issues of managing laxity and forcing migration have already been exploited in the $O(\log m)$-competitive algorithm. Therefore, we might need to involve a different flavor of strategy, such as randomization, in order to achieve a better upper bound.

References

[1] Bansal, N., Kimbrel, T., and Pruhs, K. Speed scaling to manage energy and temperature. *J. ACM*, 51(1), 2007.

[2] Chen, L., Megow, N., and Schewior, K. An $O(\log m)$-competitive algorithm for online machine minimization. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 155–163. 2016.

[3] Chen, L., Megow, N., and Schewior, K. The Power of Migration in Online Machine Minimization. In *Proceeding SPAA '16 Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 175–184. 2016.

[4] Chen, L., Megow, N., Müller, B., and Schewior, K. New Results on Online Resource Minimization. *CoRR*, abs/1407.7998, 2014.

[5] Chrobak, M., and Menyon-Mathieu, C. SIGACT news online algorithms Column 10: Competitiveness via doubling. *CSIGACT News*, 37(4):115-126, 2006.

[6] Devanur, N. R., Makarychev, K., Panigrahi, D., and Yaroslavtsev, G. Online Algorithms for Machine Minimization. *CoRR*, abs/1403.0486, 2014.

[7] Phillips, C. A., Stein, C., Torng, E., and Wein, J. Optimal time-critical scheduling via resource augmentation. In *Proc. of STOC*, page 140–149. 1997.

[8] Saha, B. Renting a cloud. In *Proc. of FSTTCS*, pages 437–448. 2013.