Maximizing Job Completions Online

Bala Kalyanasundaram[*] *Kirk R. Pruhs*[†]

Abstract

We consider the problem of maximizing the number of jobs completed by their deadline in an online single processor system where the jobs are preemptable and have release times. So in the standard three field scheduling notation, this is the online version of the problem $1 \mid r_i; pmtn \mid \sum(1 - U_i)$. We present a deterministic algorithm LAX, and show that for every instance $\mathcal{I}$, it is the case that either LAX, or the well-known deterministic algorithm SRPT (Shortest Remaining Processing Time), is constant competitive on $\mathcal{I}$. An immediate consequence of this result is a constant competitive randomized algorithm for this problem. It is known that no constant competitive deterministic algorithm exists for this problem. This is the first time that this phenomenon, the randomized competitive ratio is constant in spite of the fact that the deterministic competitive ratio is nonconstant, has been demonstrated to occur in a natural online problem. This result is also a first step toward determining how an online scheduler can use additional processors in a real-time setting to achieve competitiveness.

1 Introduction

We consider the problem of maximizing the number of jobs completed by their deadline in an online single processor system where the jobs are preemptable and have release times. So in the standard three field scheduling notation [4], this is the online version of the problem $1 \mid r_i; pmtn \mid \sum(1 - U_i)$. We present a deterministic algorithm LAX, and show that for every instance $\mathcal{I}$, it is the case that either LAX, or the well-known deterministic algorithm SRPT (Shortest Remaining Processing Time), is constant competitive on $\mathcal{I}$.

The first consequence of this result is that one can then easily obtain a constant competitive randomized online algorithm by running each of SRPT and LAX with equal probability. This assumes an oblivious adversary, that is, the adversary may not modify $\mathcal{I}$ in response to the outcome of a random event in A (for further information see one of [3, 6]). It is known that no constant competitive deterministic algorithm exists for this problem [1]. This is the first time that this phenomenon, the randomized competitive ratio is constant in spite

[*]Dept. of Computer Science, University of Pittsburgh. E-Mail: kalyan@cs.pitt.edu. Supported in part by NSF grant CCR-9734927.

[†]Dept. of Computer Science, University of Pittsburgh. E-Mail: kirk@cs.pitt.edu. Supported in part by NSF grant CCR-9734927.

of the fact that the deterministic competitive ratio is nonconstant, has been demonstrated to occur in an unarguably natural online problem. Previously, the most natural problems that demonstrably exhibited this phenomenon were various scheduling problems where the input was restricted in various ways, e.g. assuming that jobs have only two possible lengths (see for example [9]).

The second consequence of this result is that it is possible for an online deterministic scheduler, equipped with two unit speed processors, to be constant competitive with an adversary equipped with one unit speed processor. For a discussion of the usefulness of resource augmentation analysis in online scheduling problems see [7] and [10]. Broadly speaking, [7] and [10] show that an online scheduler, equipped with either faster processors or more processors, can be constant competitive with respect to flow time, and that an online scheduler, equipped with faster processors, can be constant competitive in various real-time scheduling problems. The obvious unanswered question inherent in these papers is, "How can an online scheduler make use of extra processors in real-time scheduling problems to achieve competitiveness?" The results in this paper are a first step toward answering this question.

1.1 Problem Definition

We consider the online version of the problem $1 \mid r_i; pmtn \mid \sum(1 - U_i)$. An instance $\mathcal{I}$ consists of a collection $J_1, \ldots J_n$ of jobs. Each job J_i has a *release time* r_i, a *length* or *execution time* x_i, and a *deadline* d_i. The online scheduler is unaware of J_i until time r_i, at which time the scheduler additionally learns x_i, and d_i. The scheduler may schedule J_i on a single processor for up to x_i units of time during $[r_i, d_i]$. If J_i is scheduled for x_i units of time then J_i is *completed.* The system is *preemptive*, that is, the processor may instantaneously abandon a job J_i, and later restart J_i from the point of suspension. (In a non-preemptive system, the scheduler must run a job to completion once it has started that job.) The objective function is the total number jobs completed, and our goal is to maximize this objective function.

The *laxity* ℓ_i of a job J_i is $d_i - r_i - x_i$. That is, if J_i is going to be completed, then the laxity is the amount of time that J_i will not be run between its release time and its deadline.

If S is a schedule, let $|S|$ be the number of jobs completed in S. If A is a scheduling algorithm, we let $\mathrm{A}(\mathcal{K})$ denote the schedule produced by A on input $\mathcal{K}$. We let $\textsc{Opt}(\mathcal{K})$ denote the optimal schedule on an instance $\mathcal{K}$.

In this paper we use competitiveness in several different ways. A schedule S is c-competitive with another schedule T if $|T| \leq c|S|$. A scheduling algorithm A is c-competitive if $\mathrm{A}(\mathcal{K})$ is c-competitive with $\textsc{Opt}(\mathcal{K})$ for all inputs $\mathcal{K}$. A scheduling algorithm A is *competitive* if there exists some constant c such that A is c-competitive. The *competitive ratio* of a scheduling algorithm A is the minimum c, such that A is c-competitive. The competitive ratio can be viewed as the payoff to a game played between two players, the online player and the adversary. The adversary specifies the input and services that input optimally.

1.2 Related Results

We start with results on the online version of $1 \mid r_i; pmtn \mid \sum(1 - U_i)$. Every deterministic algorithm for this problem has a competitive ratio of $\Omega(\frac{\log \Delta}{\log \log \Delta})$, where Δ is the ratio of the length of the longest job to the length of the shortest job [1]. It is easy to see from the results in this paper that the algorithm SRPT is $\Theta(\log \Delta)$ competitive. Constant competitive deterministic algorithms for special instances (e.g. equal job lengths or monotone deadlines) can also be found in [1]. If all the jobs can be completed by their deadline then the EDF (Earliest Deadline First) algorithm will produce an optimal schedule [4]. In [7] it is shown that if the online scheduler is given a faster processor than the adversary, then there is a relatively simple algorithm that is constant competitive. Further this result holds even for the more general problem $1 \mid r_i; pmtn \mid \sum(1 - U_i)w_i$, where the the jobs have associated positive benefits, and the goal is to maximize the aggregate benefit of the jobs completed by their deadline. There is no constant competitive deterministic or randomized algorithm for $1 \mid r_i; pmtn \mid \sum(1 - U_i)w_i$ [8].

We now survey results on problems that are one "change" away from the problem that we consider. For a recent general survey of online scheduling see [11].

We first consider problems where one changes the objective function. If the objective function is to minimize the number of jobs that miss their deadline, then there is no constant competitive randomized online algorithm [5]. If the objective function is to maximize processor utilization (the fraction of time that the processor is working on a job that it will complete by its deadline), then there is a 4-competitive deterministic algorithm, and this is optimal for deterministic online algorithms [2, 12]. It is well known that the algorithm SRPT minimizes the total flow time, which is the sum over all jobs of the completion time minus the release time of that job.

If one changes the job environment to disallow preemption then it is easy to see that no constant competitive randomized algorithm exists.

In [10] several results are presented in the case that the machine environment has multiple processors. In particular, they show that the algorithm that runs the job with least laxity first, and the algorithm that runs the job with the earliest deadline first, will complete all the jobs by their deadline, on instances where the adversary completes all the jobs, if they are equipped with a processor twice as fast as the adversary's processor.

The offline version of of $1 \mid r_i; pmtn \mid \sum(1 - U_i)$ can be solved in polynomial time using a dynamic programming algorithm [4].

1.3 Basic Definitions

Throughout this paper, $\mathcal{I}$ will denote a generic instance of $1|r_i, pmtn|\sum(1-U_i)$. A *schedule* S is a partial function that maps each time $t \geq 0$ to a job J_i, with $r_i \leq t \leq d_i$, that S is running at time t. Furthermore, a schedule may not run any job J_i for more than x_i time units. If S is undefined at time t then S is not running a job at time t. A schedule S *completes* a job J_i, or equivalently $J_i \in S$, if J_i is run for x_i time units. A schedule S is *resolute* if S completes every job that it runs. We say that $S \subseteq T$ if every job completed in S is completed in T. If S runs a job J_i then we denote the last time that J_i is run as $c_i(S)$. If S completes a job J_i then $c_i(S)$ is the *completion time* of J_i. If S runs a job J_i then we denote the first time that J_i is run as $f_i(S)$. A job J_i completed in a schedule S is *idle* if the processor is idle for some non-zero amount of time between when J_i is first run and when J_i is completed. We say a schedule S is *efficient* if it does not idle any jobs. We say that a job J_i is *skinny* if $x_i \geq \ell_i$, and otherwise J_i is *fat*. We define the *value* v_i of the job J_i to be $\min(x_i, \ell_i)$.

A schedule S is a *forest* if whenever $f_i(S) < f_j(S)$ then J_i is not run during the time interval $(f_j(S), c_j(S))$. If a schedule S is a forest, then we think of it as a forest of rooted trees, in the graph theoretic sense. The descendents of a job $J_i \in S$ are those jobs that are run after J_i is first run for a non-zero amount of time, but before J_i is last run. We order the children of a job in S chronologically by the time of first execution of these children. In particular, the *first child* J_j of a job J_i is the job, other than J_i, run during time $(f_i(S), c_i(S))$ that minimizes $f_j(S)$, and the *last child* J_k of J_i is the job, other than J_i, run during time $(f_i(S), c_i(S))$ that maximizes $c_j(S)$. We say a forest is *leafy* if at least half of the nodes in the forest are leaves. We define what we call *z-descendents* inductively. The job J_i is a 0-descendent of itself. If J_k is child of J_j, and J_j is a $(z-1)$-descendent of J_i, then J_k is a z-descendent of J_i. Similarly, a job J_i is the 0-ancestor of itself. If J_k is the parent of a job J_i, and J_i is the $(z-1)$-ancestor of a job J_j, then J_k is the z-ancestor of J_i.

A job J_i in a forest S is a *progenitor* if there exists an integer $z > 0$ such that J_i has 2^z z-descendents in S.

2 Algorithm Descriptions

2.1 Shortest Remaining Processing Time

Among all jobs that can be completed by their deadline, SRPT is always running the job that it can complete first. Hence, SRPT only switches jobs when it completes its current job, or when a new job J_i is released with x_i less than the remaining processing time of the job that SRPT is running.

2.2 Lax

Intuitively, Lax is trying to run jobs with high laxity. A job J_i with high laxity might be a good choice in some situations because J_i can preempted for a longer period time, without using up its laxity, than other jobs. Intuitively, we think of the value of a job as its laxity, and think of laxity as being a valuable scarce resource for each job. However, formally we seem to be required to set the value of a job J_i to be $\min(\ell_i, x_i)$, instead of the more obvious ℓ_i, because the value of high laxity to the scheduler diminishes after $\ell_i = \Omega(x_i)$. Since it was already known that a constant competitive algorithm exists in the case of all fat jobs [7], it seems that the hard case is if all jobs are skinny, that is $v_i = \ell_i$ for each job.

Lax partitions all jobs into three disjoint groups, a stack H of jobs that Lax intends to run, a collection V of jobs that Lax might consider adding to H, and the remaining jobs that are neither in H nor V. We denote the number of jobs in H by k, and the jobs in H, from bottom to top, as $J_{h(1)}, \ldots, J_{h(k)}$. Lax is always running the job $J_{h(k)}$ that is at the top of H. Let $\alpha \geq 24$ be some constant that we define later. Let $x_i(t)$ be the length of J_i not executed by Lax before time t; Or in other words, the remaining processing time of J_i at time t. A job J_i is *viable* at time t if $r_i \leq t$ and $d_i - t - x_i(t) \geq \ell_i/2$. Lax maintains a collection V of jobs J_i

- that are not currently in H, and that were never previously in H,
- that are currently viable, and
- that satisfy $\alpha x_i \leq v_{h(k)}$.

Intuitively, when Lax considers adding a job to its stack H, it only considers jobs in V. Jobs that are no longer viable have used up too much of their original laxity. Jobs J_i with $\alpha x_i > v_{h(k)}$ will use up too much of the laxity of the job $J_{h(k)}$. We exclude jobs that have been in H previously largely because it simplifies our analysis. For each job J_i, let $\mathrm{Push}(i)$ be the time that J_i was pushed on H, and $\mathrm{Pop}(i)$ be the time that J_i was popped off H. If J_i was never pushed onto H then $\mathrm{Push}(i) = \mathrm{Pop}(i) = +\infty$.

There are two type of events that might cause Lax to change H, the release of a job, and completion of the top job in H. The algorithms for both of these events use the following procedure `Fill` that repeatedly pushes the highest value job in V onto H.

```
        Procedure Fill
        While V is not empty Do
               Select the job J_j ∈ V with maximum value
(1)            Push J_j onto H
```

Note that each push and each pop in the code can potentially change V. Now consider a time r_i at which a job J_i is released. Intuitively, if J_i won't use up too much of the laxity of the top job in H then J_i should be pushed onto H. Otherwise, the status quo is acceptable unless J_i is more valuable than the job at the top of the stack H, and J_i doesn't use up too much of the laxity of the job that is second from the top of H. In this case, the stack H is popped and the procedure `Fill` is called. The job J_i doesn't simply replace $J_{h(k)}$ because, in order to make our analysis work, we seem to need the invariant that `Fill` is called every time that a job is added to H. More formally, LAX executes the following code when J_i is released:

```
        If v_h(k) ≥ αx_i Then
(2)         Push J_i onto H
        Else If v_h(k-1) ≥ αx_i and v_i > v_h(k) Then
(3)         Pop H
            Select the job J_j ∈ V with maximum value
(4)         Push J_j onto H
            Call Fill
```

The first loop in `Fill` is unrolled before the call to `Fill` because we will have to treat the push in line (4) differently in our analysis than the push in line (1) of `Fill`. Note that $J_i \in V$ after the pop in line (3).

LAX responds to the completion of the top job $J_{h(k)}$ in H at time t by popping jobs off the stack that can not possibly complete by their deadline, and then calling `Fill`. More formally, LAX runs the following code when it completes the top job of H:

```
(5)     Pop H
        While t + x_h(k)(t) > d_h(k) Do
(6)         Pop H
        Call Fill
```

A forest S is *shrinking* if whenever J_j is a descendent of J_i in S it is the case that $\alpha x_j \leq v_i$. It is obvious that LAX$(\mathcal{I})$ is an efficient forest. Lemma 1 also shows that LAX$(\mathcal{I})$ is shrinking.

Lemma 1 At all times throughout the execution of LAX it is the case that $v_{h(i)} \geq \alpha x_{h(i+1)}$, $v_{h(i)} \geq \alpha v_{h(i+1)}$, and $x_{h(i)} \geq \alpha x_{h(i+1)}$, for $1 \leq i \leq k-1$.

Proof: That $v_{h(i)} \geq \alpha x_{h(i+1)}$ follows since $J_{h(i+1)}$ was in V at the time that it was pushed onto H. That $v_{h(i)} \geq \alpha v_{h(i+1)}$ follows because $x_{h(i+1)} \geq v_{h(i+1)}$. That $x_{h(i)} \geq \alpha x_{h(i+1)}$ follows because $x_{h(i)} \geq v_{h(i)}$. ■

3 Algorithm Analysis

We prove that for all instances $\mathcal{I}$, $|\text{Lax}(\mathcal{I})| + |\text{SRPT}(\mathcal{I})| = \Omega(|\text{Opt}(\mathcal{I})|)$. To accomplish this we will want to compare $\text{SRPT}(\mathcal{I})$ and $\text{Lax}(\mathcal{I})$ with $\text{Opt}(\mathcal{I})$. Unfortunately, structurally $\text{Opt}(\mathcal{I})$ can be quite different from $\text{Lax}(\mathcal{I})$, thus making a direct comparison difficult. Intuitively, the main purpose of section 3.1 is to show that there are near optimal schedules that are structurally similar to the schedules produced by Lax, thus facilitating our comparison. To be more specific, we need the following definitions.

Definition 2 We define the *pseudo-release time*, denoted pr_i, of a job J_i. If J_i is in H at some time then $pr_i = \text{Push}(i)$. Else let $t \geq r_i$ be the earliest time such that $\alpha x_{h(k)} > v_i$ after Lax has completely responded to all events at time t. If $t \leq r_i + \ell_i/2$ then $pr_i = t$, else $pr_i = +\infty$. Let $\mathcal{J}$ be the collection of jobs in $\mathcal{I}$ with finite pseudo-release times, and let $\mathcal{N}$ be the collection of jobs with infinite pseudo-release times. So $\mathcal{I} = \mathcal{N} \cup \mathcal{J}$.

The intuition behind the definition of pseudo-release time is that pr_i is the first time that Lax reasonably had a chance to consider running J_i. In section 3.1, we construct a forest COpt that is competitive with $\text{Opt}(\mathcal{N})$. It is a combinatorial fact that for every forest at least half the nodes are leaves or first children. We argue that $\text{SRPT}(\mathcal{I})$ is competitive with both the number of leaves in COpt, and with the number of first children in COpt. The latter argument is more involved, and requires the construction of an auxiliary shrinking forest DOpt of the first children in COpt. Hence, we can conclude that the adversary can't earn too many credits from jobs in $\mathcal{N}$, and therefore it is sufficient to show that Lax or SRPT is competitive with $\text{Opt}(\mathcal{J})$.

In section 3.1 we construct a forest BOpt that is competitive with $\text{Opt}(\mathcal{J})$. Once again its easy to see that $\text{SRPT}(\mathcal{I})$ is competitive with the number of leaves in BOpt. So we are left to show that SRPT and Lax are competitive with the number of first children in BOpt. To accomplish this we will construct an auxiliary forest AOpt such that

- AOpt is shrinking, and
- each job $J_i \in$ AOpt it is the case that $f_i(\text{AOpt}) = pr_i$.

The shrinking property is required so that the jobs in ancestor-to-descendent paths in AOpt might all legally be in Lax's stack at the same time. The two properties together allow us to show in section 3.4 the keystone result of the paper, that is, during any period of time during which AOpt started executing four different jobs, it must be the case that Lax executed a push in either line (1) or line (2) of its code. We show in the section 3.3 that the number of jobs that Lax and SRPT complete is at least a constant fraction of the number of times that Lax executes the pushes in lines (1) and (2) of its code. In section 3.5, we pull all of the preliminary results together to get the final result that either Lax or SRPT is constant competitive.

Name	Where Defined	Comments
EDF	Lemma 8	Makes schedule a resolute forest
Shrink	Lemma 9	Makes schedule shrinking
Late	Lemma 10	Makes $c_i = d_i$ for last children
Early	Lemma 11	Makes $f_i = r_i$ for first children
Back	Lemma 12	Makes $f_i = pr_i$ for first children
Klcl	Lemma 13	Keeps last children and leaves
Kfcl	Lemma 14	Keeps first children and leaves
CFill	Lemma 15	Makes schedule efficient without changing completions times
RFill	Lemma 16	Makes schedule efficient without changing first run times
Dmany	Lemma 17	Removes jobs that have too many descendents
Dl	Definition 18	Removes leaves
AOpt	Definition 18	Approximation to $\text{Opt}(\mathcal{J})$ if there are few leaves
BOpt	Definition 18	Approximation to $\text{Opt}(\mathcal{J})$ if there are many leaves
DOpt	Definition 21	Approximation to $\text{Opt}(\mathcal{N})$ if there are few leaves
COpt	Definition 21	Approximation to $\text{Opt}(\mathcal{N})$ if there are many leaves

Table 1: Transformations and schedules

3.1 Structure of the Optimal Schedule

In this subsection, we construct many schedules, and many transformations between schedules. The reader may find table 1 helpful in keeping track of these schedules and transformations.

3.1.1 Preliminaries

We start with some preliminary definitions and and lemmas. Let Z be a function from forest schedules to forest schedules. We say that Z is *descendent conserving* if for all schedules S it is that case that every job completed in $Z(S)$ is completed in S, and for every job $J_i \in Z(S)$, if J_i is a descendant of J_j in $Z(S)$ then J_i is a descendant of J_j in S. Intuitively, a transformation is descendent conserving if no job gains a descendent.

Lemma 3 Let Z be a descendent conserving transformation. Then $Z(T)$ is shrinking whenever T is shrinking.

Proof: This is immediate since a descendent conserving relation adds no new descendents. ■

Lemma 4 Any transformation from schedules to schedules that only deletes jobs, without changing when the remaining jobs are run, is descendent conserving.

Lemma 5 Let S be a shrinking schedule. If $z \geq 1$, and J_j is a z-descendent of J_i in S, then $\alpha^z x_j \leq v_i$.

Proof: This follows easily by induction on z using the definition of shrinking. ■

Lemma 6 The total executions times of the descendents of any non-progenitor J_i in a shrinking resolute forest S is at most $v_i/4$.

Proof: Since S is shrinking, each z-descendent, $z > 0$, J_j of J_i satisfies $x_j \leq v_i/\alpha^z$ by lemma 5. Since J_i is not a progenitor in S, the total execution time of the descendents of J_i is S is at most

$$\sum_{z=1}^{\infty} 2^z \frac{v_i}{\alpha^z} = v_i \frac{\frac{2}{\alpha}}{1-\frac{2}{\alpha}} = \frac{2v_i}{\alpha - 2} \leq \frac{v_i}{4}$$

The last inequality follows since $\alpha \geq 10$. ■

Lemma 7 In any resolute forest schedule S at least half of the jobs are non-progenitors.

Proof: Consider the following amortization scheme. Each non-progenitor J_i is initially given one credit. Then each non-progenitor J_i contributes $1/2^{y+1}$ credits to its y-ancestor in S, $y \geq 0$. Note that the total contribution made by J_i is less than one. We now argue by contradiction that each job $J_j \in S$ gets an aggregate contribution of at least 1/2 credits. To reach a contradiction, let $J_j \in S$ be a job that receives less than 1/2 credits, and furthermore, all of the descendents of $J_j \in S$ received at least 1/2 credit. The job J_j must be a progenitor, or it would have $\frac{1}{2}$ credit left from its original allotment. By definition of a progenitor, there must exist a $z \geq 1$ such that J_j has a collection $\mathcal{Z}$ of 2^z z-descendents that each received at least 1/2 a credit. Let $J_i \in \mathcal{Z}$, and let J_k be a descendent of J_i that contributed c credits to J_i. Then J_k will contribute at least $\frac{c}{2^z}$ credits to J_j. Hence, the total contribution that the descendents of J_i make to J_j is at least $\frac{1}{2^{z+1}}$. Hence, J_j must receive a contribution of at least $\frac{1}{2}$, which is a contradiction. ■

3.1.2 The Transformations

We define the various schedule transformations that will be used to construct AOPT, BOPT, COPT, and DOPT. We start by showing how to turn an arbitrary schedule into an efficient resolute forest.

Lemma 8 For every schedule T, there is an efficient resolute forest $\mathrm{EDF}(T)$, that completes exactly the same jobs as T.

Proof: It is well known that if there is a feasible schedule that completes all of the jobs, then the Earliest Deadline First (EDF) schedule, which maintains the invariant that it always runs the uncompleted job with earliest deadline, also completes all the jobs [4]. Let EDF(T) be the EDF schedule of the jobs completed in T. The result then follows immediately. ■

We now show how to transform an arbitrary resolute forest into a shrinking resolute forest.

Lemma 9 Let $\beta = \frac{\alpha-3}{5\alpha(\alpha-1)}$. For every resolute forest T, there is another resolute forest $\textsc{Shrink}(T)$ such that

1. $\textsc{Shrink}(T) \subseteq T$,
2. $\textsc{Shrink}(T)$ is shrinking,
3. $|\textsc{Shrink}(T)| \geq \frac{\beta}{4}|T|$.

Proof: We break the proof into cases. In the first case assume that there are more skinny jobs in T than there are fat jobs. Let A be T with the fat jobs deleted. Note $|A| \geq |T|/2$. Process each tree in A from the root to the leaves in a pre-order fashion. At each node J_i, delete the descendants J_j of J_i with $\alpha x_j \geq \ell_i$. Note that for each J_i it is the case that at most α jobs will be deleted. Call the resulting forest C. Note that $|C| \geq \frac{|A|}{\alpha+1} \geq \frac{|T|}{2(\alpha+1)}$. Furthermore, C is shrinking because the construction of C guarantees that if J_j is a descendent of J_i in C then $\alpha x_j \leq \ell_i$, and hence $\alpha x_j \leq v_i$ since each job in C is skinny.

In the second case assume that there are more fat jobs in T than there are skinny jobs. Let A be T with the skinny jobs deleted. Hence $|A| \geq |T|/2$. In [7], the online version of the problem $1|r_i, pmtn|\sum w_i(1-U_i)$ (maximizing the aggregate benefit of the jobs completed by their deadline) is considered, and an algorithm $\textsc{Slacker}$ is presented. It is shown in [7] that, under the assumption that every job is fat, the competitive ratio of $\textsc{Slacker}$ is at most $\frac{5c(c-1)}{c-3}$, where c is a parameter of $\textsc{Slacker}$ satisfying $c > 3$. The schedule produced by $\textsc{Slacker}$ is a forest. Furthermore, if each job has unit benefit (as is the case here) the schedule produced by $\textsc{Slacker}$ has the property that if J_j is a descendent of J_i then $\lfloor \log_c \frac{1}{x_i} \rfloor < \lfloor \log_c \frac{1}{x_j} \rfloor$. Consider the schedule produced by $\textsc{Slacker}$, with $c = \alpha$, when the input is the jobs completed in A. Delete from this schedule any jobs not completed by $\textsc{Slacker}$, and call the resulting schedule B. Hence, $|B| \geq \frac{|A|(\alpha-3)}{5\alpha(\alpha-1)} = \beta \cdot |A|$. We consider two subcases. In the first subcase, assume that there are more jobs in B with odd depth. In this case, let C be the schedule formed from B by deleting the jobs with even depth. In the second subcase, assume that there are more jobs in B with even depth. In this case, let C be the schedule formed from B by deleting the jobs with odd depth. Hence, $|C| \geq |B|/2$. If J_j is a descendent of J_i in C, then $1 + \lfloor \log_c \frac{1}{x_i} \rfloor < \lfloor \log_c \frac{1}{x_j} \rfloor$ because there used to be at least one intermediate node between J_i and J_j in B. Therefore $1 + \log_c \frac{1}{x_i} < \log_c \frac{1}{x_j}$, and hence, $\alpha x_j < x_i$. Therefore C is shrinking since each job in A is fat. Also $|C| \geq \frac{\beta}{4}|T|$.

Hence, in either case $|C| \geq \frac{\beta}{4}|T|$, since $\frac{\beta}{4} \leq \frac{1}{2(\alpha+1)}$ for $\alpha > 3$. In either case, we set $\textsc{Shrink}(T) = C$. ■

The next three lemmas show how to move the completion of last children back to their deadlines, and how to move the first run times of first children up to either their release time, or pseudo-release time.

Lemma 10 For every resolute forest schedule T, there is another resolute forest schedule $\text{LATE}(T)$ that completes exactly the jobs in T and such that

1. for all last children $J_i \in \text{LATE}(T)$ it is the case that $c_i(\text{LATE}(T)) = d_i$,
2. if T is efficient then $\text{LATE}(T)$ is efficient, and
3. LATE is descendent conserving.

Proof: Let $S_0 = T$. We then repeatedly apply the following construction to the schedule S_a, $a \geq 0$, to create a schedule S_{a+1}, until it is no longer possible to do so. Let J_j be a last child of a job J_i in S_a such that $c_j(S_a) \neq d_j$. Let $m = \min(d_j, c_i(S_a))$. Let P be the period of time, during the time period $[f_j(S_a), m]$, that S_a is running either J_j or J_i. Let y be the total time that S_a is running J_i during time P. To obtain S_{a+1} we modify the schedule S_a to run J_i for the first y time units in P, and J_j for the last x_j time units in P. Note that this is a legal schedule because, J_i is not run before its release time since $r_i \leq f_i(S_a) < f_j(S_a)$, and J_j is not after its deadline since $d_j \leq m$.

If $d_j \geq c_i(S_a)$ then J_j becomes a sibling of J_i. In this case, the aggregate depths of the nodes in S_{a+1} is strictly less than the aggregate depths of the nodes in S_{a+1}, and the number of last children in S_{a+1} that complete at their deadline is the same as the number of last children in S_a that complete at their deadline.

Otherwise, if $d_j < c_i(S_a)$ then J_j stays a child of J_i. Furthermore, in some cases, some children of J_j may become children of J_i. In this case, the aggregate depths of the nodes in S_{a+1} is not larger than the aggregate depths of the nodes in S_{a+1}, and the number of last children in S_{a+1} that complete at their deadline is strictly larger than the number of last children in S_a that complete at their deadline.

Hence, this construction must terminate at some schedule, which we define to be $\text{LATE}(T)$. Condition 1 holds for $\text{LATE}(T)$ since the construction is no longer applicable. Condition 2 holds since the above transformation does not create any new idle times. Condition 3 holds since the construction creates no new descendents. ■

The proof of the following two lemmas follow from the proof of the previous lemma by symmetry.

Lemma 11 For every resolute forest schedule T, there is another resolute forest schedule $\text{EARLY}(T)$ that completes exactly the same jobs as T and such that

1. for all first children $J_i \in \text{EARLY}(T)$ it is the case that $f_i(\text{EARLY}(T)) = r_i$,

2. if T is efficient then $\text{EARLY}(T)$ is efficient, and

3. EARLY is descendent conserving.

Lemma 12 Let T be a resolute forest schedule such that for every job $J_i \in T$ it is the case that $f_i(T) \geq pr_i$. Then there is another resolute forest schedule $\text{BACK}(T)$ that completes exactly the same jobs as T, and such that

1. for all first children $J_i \in \text{BACK}(T)$ it is the case that $f_i(\text{BACK}(T)) = pr_i$,

2. if T is efficient then $\text{BACK}(T)$ is efficient, and

3. BACK is descendent conserving.

We now show that a lot of jobs in any forest must be either leaves or last children, and that a lot of jobs in any forest must be either leaves or first children. We need this lemma because we only know how to show that LAX and SRPT are competitive with the number of leaves, first children, or last children in a particular schedule.

Lemma 13 Let $\text{KLCL}(S)$ be the schedule obtained from the resolute forest schedule S by deleting all jobs that are not last children or leaves. Then $\text{KLCL}(S)$ is a resolute forest and $|\text{KLCL}(S)| \geq |S|/2$.

Proof: We prove the cardinality claim separately for each tree T in the forest. We use induction on the the height of T. The claim follows immediately if T has height 1, and hence consists of a single node. Otherwise, let r be the root of T, let $c_1, \ldots, c_m$ be the children of r in T, let T_i be the subtree of T rooted at c_i, $1 \leq i \leq m$, and let $L(A)$ be the number of leaves plus the number of last children in a tree A. Then the claim follows since

$$L(T) = 1 + \sum_{i=1}^{m} L(T_i) \geq 1 + \sum_{i=1}^{m} |T_i|/2 = 1 + (|T| - 1)/2 \geq |T|/2$$

The first inequality is by induction. ■

The proof of lemma 14 follows from the proof of lemma 13 by symmetry.

Lemma 14 Let $\text{KFCL}(S)$ be the schedule obtained from the resolute forest schedule S by deleting all jobs that are not first children or leaves. Then $\text{KFCL}(S)$ is a resolute forest and $|\text{KFCL}(S)| \geq |S|/2$.

The new two lemmas show how to make a resolute forest efficient without changing the completion times, and first run times, respectively.

Lemma 15 For every resolute forest schedule T there is another resolute forest schedule $\textsc{CFill}(T)$ that completes exactly the same jobs as T and such that

1. $\textsc{CFill}(T)$ is efficient,
2. for all $J_i \in T$, $c_i(\textsc{CFill}(T)) = c_i(T)$, and
3. $\textsc{CFill}$ is descendent conserving.

Proof: We then repeatedly apply the following construction to T to obtain $\textsc{CFill}(T)$. Let J_i be an idle job with no idle descendents. Then the initial execution of J_i is delayed for as long as possible so as to still allow J_i to complete at its original completion time, provided that it is always run during the intervening idle times. ■

Lemma 16 For every resolute forest schedule T there is another resolute forest schedule $\textsc{RFill}(T)$ that completes exactly the same jobs as T and such that

1. $\textsc{RFill}(T)$ is efficient,
2. for all $J_i \in T$, $f_i(\textsc{RFill}(T)) = f_i(T)$, and
3. $\textsc{RFill}$ is descendent conserving.

Proof: We then repeatedly apply the following construction to T to obtain $\textsc{RFill}(T)$. Let J_i be an idle job with no idle descendents. Then J_i is run during each idle time, after J_i is first run, until J_i completes. ■

We now show how to modify any resolute shrinking forest so that the sum of the execution times of the descendents of each job J_i is small. This will allow us the freedom to either move forward or move back the execution of J_i.

Lemma 17 Let S be an arbitrary shrinking resolute forest schedule. Let $\textsc{DMany}(S)$ be the schedule derivable from S by deleting each progenitor in S. Then $\textsc{DMany}(S)$ is a shrinking resolute forest schedule and

1. the sum of the execution times of the descendents of each $J_i \in \textsc{DMany}(S)$ is at most $v_i/4$.
2. $|\textsc{DMany}(S)| \geq |S|/2$.

Proof: Condition 1 follows from lemma 6. Condition 2 follows from lemma 7. ■

3.1.3 The Approximate Optimal Schedules

We are now ready to define AOpt, BOpt, COpt, and DOpt.

Definition 18 Let $\mathrm{DL}(S)$ be the resolute forest obtained from a resolute forest S by deleting the leaves in S. Let

$$\mathrm{BOPT} = \mathrm{KLCL}(\mathrm{LATE}(\mathrm{SHRINK}(\mathrm{EDF}(\mathrm{OPT}(\mathcal{J})))))$$

and

$$\mathrm{AOPT} = \mathrm{BACK}(\mathrm{CFILL}(\mathrm{DMANY}(\mathrm{DL}(\mathrm{BOPT}))))$$

Lemma 19 BOpt is resolute shrinking forest schedule with

$$|\mathrm{BOPT}| \geq \frac{\beta}{8}|\mathrm{OPT}(\mathcal{J})|$$

Proof: $\mathrm{EDF}(\mathrm{OPT}(\mathcal{J}))$ is a resolute forest, and all of the later transformations preserve resoluteness and forestness. The schedule $\mathrm{SHRINK}(\mathrm{EDF}(\mathrm{OPT}(\mathcal{J})))$ is shrinking, and all of the later transformations are descendent conserving. Hence, BOpt is shrinking. That

$$|\mathrm{BOPT}| \geq \frac{\beta}{8}|\mathrm{OPT}(\mathcal{J})|$$

follows from lemmas 9, 10, and 13. ∎

Lemma 20 AOpt is a shrinking resolute efficient forest schedule such that

1. for all $J_i \in \mathrm{AOPT}$, it is the case that $f_i(\mathrm{AOPT}) = pr_i$, and
2. if BOpt is not leafy, then $|\mathrm{AOPT}| \geq \frac{\beta}{32}|\mathrm{OPT}(\mathcal{J})|$.

Proof: All of the transformations involved preserve resoluteness and forestness. AOpt is shrinking by lemma 9, and since, as we have shown in lemmas 4, 15 and 12, the transformations Dl, CFill, Dmany, and Back are descendent conserving. Condition 1 follows by lemma 12. By lemma 19,

$$|\mathrm{BOPT}| \geq \frac{\beta}{8}|\mathrm{OPT}(\mathcal{J})|$$

Suppose BOpt is not leafy. Then

$$|\mathrm{DL}(\mathrm{BOPT})| \geq \frac{\beta}{16}|\mathrm{OPT}(\mathcal{J})|$$

Let

$$A = \mathrm{CFILL}(\mathrm{DMANY}(\mathrm{DL}(\mathrm{BOPT})))$$

By lemmas 15 and 17,

$$|A| \geq \frac{\beta}{32}|\textsc{Opt}(\mathcal{J})|$$

Let $J_i \in A$. Then by lemma 17 $c_i(A) = d_i$. Also $f_i(A) \geq d_i - (x_i + v_i/4)$, or J_i would have been deleted by DMANY. Hence, since $d_i - x_i = r_i + \ell_i$, $f_i(A) \geq r_i + \ell_i - v_i/4 \geq r_i + \ell_i/2$. and $f_i(A) \geq pr_i$ since $pr_i \leq r_i + \ell_i/2$ for $J_i \in \mathcal{J}$. Therefore, BACK is applicable to A. Finally,

$$|\textsc{Back}(A)| = |\textsc{AOpt}| \geq \frac{\beta}{32}|\textsc{Opt}(\mathcal{J})|$$

by lemma 12. ∎

Definition 21 Let

$$\textsc{COpt} = \textsc{Kfcl}(\textsc{Early}(\textsc{Shrink}(\textsc{EDF}(\textsc{Opt}(\mathcal{N})))))$$

and

$$\textsc{DOpt} = \textsc{RFill}(\textsc{Dl}(\textsc{COpt}))$$

Lemma 22 COPT is a resolute shrinking forest schedule with

$$|\textsc{COpt}| \geq \frac{\beta}{8}|\textsc{Opt}(\mathcal{N})|$$

Proof: COPT is resolute forest by lemma 8, and the fact that SHRINK, EARLY, and KFCL maintain resoluteness and forestness. COPT is shrinking by lemma 9, and the fact that EARLY and KFCL are descendent conserving. So

$$|\textsc{COpt}| \geq \frac{\beta}{8}|\textsc{Opt}(\mathcal{N})|$$

follows by lemmas 8, 9, 11, and 14. ∎

Lemma 23 DOPT is a shrinking resolute efficient forest schedule such that

1. for all $J_i \in \textsc{DOpt}$, it is the case that $f_i(\textsc{DOpt}) = r_i$, and
2. if COPT is not leafy, then $|\textsc{DOpt}| \geq \frac{\beta}{16}|\textsc{Opt}(\mathcal{N})|$.

Proof: That DOPT is a resolute shrinking forest schedule follows from lemma 22, and the fact that DL, and RFILL are descendent conserving, and preserve resoluteness and forestness. Condition 1 follows by lemma 11, the fact that DMANY does change first runs times, and lemma 16. Suppose that COPT is not leafy. Then

$$|\textsc{DOpt}| \geq \frac{\beta}{16}|\textsc{Opt}(\mathcal{N})|$$

follows by the definition of leafiness, lemma 22, and the fact that RFILL doesn't remove jobs. ∎

3.2 Jobs with Infinite Pseudo-Release Times

In this section we will show that SRPT is constant competitive with respect to $\text{O}\textsc{pt}(\mathcal{N})$. This will allow us subsequently restrict our attention to jobs in $\mathcal{J}$.

Lemma 24 Let S be an arbitrary resolute forest schedule for $\mathcal{K}$. Then $|\text{SRPT}(\mathcal{K})|$ is at least half the number of leaves in S.

Proof: We give an amortization scheme by which the jobs in $\text{SRPT}(\mathcal{K})$ pay for the jobs in S. Each job $J_i \in \text{SRPT}(\mathcal{K})$ is initially given one credit, and contributes half a credit to the job J_j that S is running at time $c_i(\text{SRPT}(\mathcal{K}))$ (if such a job J_j exists). Consider a job J_j that is a leaf in S. Hence, $c_j(S) = f_j(S) + x_j$. We claim that J_j ends up with half a credit in this scheme. If $J_j \in \text{SRPT}(\mathcal{K})$ it has at least half a credit left from its initial allocation. Otherwise, $\text{SRPT}(\mathcal{K})$ must complete a job during the interval $[f_j(S), f_j(S)+x_j]$ since $\text{SRPT}(\mathcal{K})$ could have opted to run J_j at time $f_j(S)$. ■

Lemma 25 $|\text{SRPT}(\mathcal{I})| \geq \frac{1}{4}|\text{DO}\textsc{pt}|$

Proof: We now give an amortization scheme by which the jobs in $\text{SRPT}(\mathcal{I})$ pay for the jobs DOpt. Each job $J_j \in \text{SRPT}(\mathcal{I})$ is initially given one credit, and contributes credits to other jobs in the following ways. The job that DOpt is running at time $c_j(\text{SRPT}(\mathcal{I}))$ is given $\frac{1}{2}$ credit in a type 1 transfer. The job that DOpt is running at time $c_j(\text{L}\textsc{ax}(\mathcal{I}))$ is given $\frac{1}{2}$ credit in a type 2 transfer.

Since it is obvious that each job $J_j \in \text{SRPT}(\mathcal{I})$ contributes at most one point, it is sufficient, by lemma 7, to show that each non-progenitor $J_i \in$ DOpt receives at least $\frac{1}{2}$ credit.

Since DOpt is a forest, during $[r_i, r_i + v_i/2]$ it must be the case that DOpt is only running J_i or descendents of J_i. Note that by lemma 23 it is the case that $f_i(\text{DO}\textsc{pt}) = r_i$. The job J_i has at most one child in DOpt since J_i is not a progenitor in DOpt. Hence, since DOpt is efficient, we can partition $[r_i, r_i + v_i/2]$ into three intervals $[r_i, a)$, (a, b) and $(b, r_i + v_i/2]$ such that during $[r_i, a)$ it must be the case that DOpt is running J_i, during (a, b) it must be the case that DOpt is running descendents of J_i, and during $(b, r_i + v_i/2]$ it must be the case that DOpt is running J_i. Note that it may be the case that J_i is a leaf, and hence $a = b$. Since J_i is not a progenitor in DOpt and DOpt is shrinking, by lemma 6 it must be the case that $b - a \leq v_i/4$. Hence, at least one of $(a - r_i)$ or $r_i + v_i/2 - b$ must be of size at least $v_i/4$.

Assume that $a - r_i \geq v_i/4$, the argument is the same if $r_i + v_i/2 - b \geq v_i/4$. By the definition of $\mathcal{N}$, at all times during the time interval $[r_i, a]$ it must be the case that Lax was always running jobs J_j with $\alpha x_j \leq v_i$. Let X_z be the collection of jobs with lengths in the range $R_z = (\frac{v_i}{\alpha^{z+1}}, \frac{v_i}{\alpha^z}]$ that Lax was running during $[r_i, a]$, let T_z be the times that these jobs were being run during $[r_i, a]$, and let $|T_z|$ the measure of T_z. There must exists a $z > 0$, such that $|X_z| \geq \frac{v_i}{4 \cdot 2^z}$. Let t be the unique point in time such that $|T_z \cap [r_i, t]| = \frac{v_i}{4 \cdot \alpha^z}$. The time t exists since $\alpha > 2$. We claim that LAX must have begun running a job $J_j \in X_z$ at some time $s \in [r_i, t]$, and $s + x_j \leq s + \frac{v_i}{4 \cdot \alpha^z} \leq a$. In order see that such an s exists, observe that $|T_z|$ over the length of the longest job in X_z is at least 3. That is,

$$\frac{|X_z|}{\frac{v_i}{\alpha^z}} \geq \frac{\frac{v_i}{4 \cdot 2^z}}{\frac{v_i}{\alpha^z}} \geq 3$$

The second inequality follows since $z \geq 1$, and $\alpha \geq 24$.

We consider two cases. In the first case, assume that SRPT didn't complete J_j before time t. In this case SRPT must complete a job J_h during $[s, s + x_j]$ since SRPT could have opted to run J_j at time s. The job J_h then contributes $\frac{1}{2}$ credits to J_i in a type 1 transfer. In the second case, assume that SRPT did complete J_j before time s. Then J_j contributes $\frac{1}{2}$ point to J_i in a type two transfer. ∎

Lemma 26 $|\mathrm{SRPT}(\mathcal{I})| \geq \frac{\beta}{64} |\mathrm{OPT}(\mathcal{N})|$.

Proof: First assume that COPT is leafy. Then $|\mathrm{SRPT}(\mathcal{I})| \geq \frac{1}{2} |\mathrm{COPT}|$ by lemma 24. Furthermore, $|\mathrm{COPT}| \geq \frac{\beta}{8} |\mathrm{OPT}(\mathcal{N})|$ by lemma 22. Hence in this case, $|\mathrm{SRPT}(\mathcal{I})| \geq \frac{\beta}{16} |\mathrm{OPT}(\mathcal{N})|$.

If COPT is not leafy then $|\mathrm{SRPT}(\mathcal{I})| \geq \frac{1}{4} |\mathrm{DOPT}|$ by lemma 25, and $|\mathrm{DOPT}| \geq \frac{\beta}{16} |\mathrm{OPT}(\mathcal{N})|$ by lemma 23. Hence in this case, $|\mathrm{SRPT}(\mathcal{I})| \geq \frac{\beta}{64} |\mathrm{OPT}(\mathcal{N})|$. ∎

3.3 Counting Pushes

Let C be the total number of times that the pushes in line (1) and line (2) of LAX were executed. Our goal in this subsection is to show that the number of jobs completed by LAX and SRPT is $\Omega(C)$. In subsequent sections this will allow us to assume that LAX and SRPT earn an amortized $\Omega(1)$ credit whenever LAX executes a push in line (1) or line (2). Let $\mathcal{D}$ be the collection of jobs that were popped in line (6) of LAX. Note that jobs in $\mathcal{D}$ are not completed by LAX. We start by giving an amortization scheme that demonstrates that the number of jobs completed by SRPT and LAX is $\Omega(|\mathcal{D}|$.

For a time t, we define t^+ to be the time just after time t such that during the period $(t, t^+]$ it is the case that LAX does not complete a job, and no job is released. For a time t, we define t^- to be the time just before time t such that during the period $[t^-, t)$ it is the case that LAX does not complete a job, and no job is released. We say that the *depth* of a job in the stack H is its depth from the top of H, more precisely, the depth of $J_{h(j)}$ is $k - j + 1$. Recall that k is the height of H.

Amortization Scheme: Each job completed by SRPT is initially given one credit, and each job completed by LAX is initially given one credit. We will distribute these credits to other jobs in the following ways. If LAX completes a job J_j at a time t, then for each integer $a \in [1, k]$ the job of depth a in H at time t^+ receives $1/2^{a+1}$ credits from J_j in a LAX transfer. If SRPT completes a job J_j at a time t, then for each integer $a \in [1, k]$ the job of depth a in H at time t receives $1/2^{a+1}$ credits from J_j in a SRPT-1 transfer. If SRPT completes a job J_j and LAX ran J_j at some point in time, then for each integer $a \in [1, k]$ the job of depth a in H at time $\text{Push}(j)$ receives $1/2^{a+1}$ credits from J_j in a SRPT-2 transfer.

Lemma 27 The number of credits transfered from each job completed by LAX or SRPT by the above amortization scheme is at most 1.

We now fix an arbitrary job $J_i \in \mathcal{D}$. We want to show that $\Omega(1)$ credits were transfered to J_i under this amortization scheme. If J_j is above J_i on the stack H at some time during $(\text{Push}(i), \text{Pop}(i))$, then we say that J_j is z *above* J_i if the depth of J_i minus the depth of J_j is z (note that both of these depths are invariant during $(\text{Push}(j), \text{Pop}(j))$). Intuitively, we show in the next lemma that there is some z such that LAX is running that are z above J_i in H for a reasonably long period of time.

Lemma 28 There exists an integer $z > 0$, such that during the period $[\text{Push}(i), \text{Pop}(i)]$, LAX was running jobs that are z above J_i for at least $\frac{\ell_i}{2^{z+1}}$ units of time.

Proof: During the period $[\text{Push}(i), \text{Pop}(i)]$ the algorithm LAX will only run J_i, and jobs above J_i in H. Since J_i was viable when it was added to H, and $J_i \in \mathcal{D}$, LAX ran jobs other than J_i for at least $\ell_i/2$ units of time. Assume to reach a contradiction that, for each $z > 0$, LAX ran jobs that are z above J_i less than $\frac{\ell_i}{2^{z+1}}$ units of time during $[\text{Push}(i), \text{Pop}(i)]$. Then the total time that LAX wasn't running J_i during $[\text{Push}(i), \text{Pop}(i)]$ would have to be less than $\sum_{z=1}^{\infty} \frac{\ell_i}{2^{z+1}} = \frac{\ell_i}{2}$. This contradicts the assumption that $J_i \in \mathcal{D}$. ∎

Let z be the integer from lemma 28. We now define a collection $I_1 = [a_1, b_1], \ldots, I_s = [a_s, b_s]$ of disjoint intervals as follows:

- $[a_u, b_u] \subseteq [\text{Push}(i), \text{Pop}(i)]$ for $1 \leq u \leq s$,
- $b_j < a_{j+1}$ for $1 \leq j < s$,
- during the interval $[a_u, b_u]$, $1 \leq u \leq s$, the algorithm LAX is running jobs that are z above J_i,
- during $(\text{Push}(i), a_1)$ and $(b_s, \text{Pop}(i))$ the algorithm LAX is never running a job that is z above J_i, and
- during (b_u, a_{u+1}), $1 \leq u < s$, the algorithm LAX is never running a job that is z above J_i.

That is, these intervals are the times when LAX is running jobs that are z above J_i. In the next three lemmas we quantify the credits earned by J_i.

Lemma 29 **During the time period (b_u, a_{u+1}), $1 \leq u < s$, there is a LAX transfer that nets J_i at least $1/2^{z+2}$ credits.**

Proof: If LAX is executing a job that is more that z above J_i at time a_{u+1}^- then let $t = a_{u+1}$. Otherwise, if during the period (b_u, a_{u+1}) it is always the case that the size of H is less than z then let $t = b_u$. Otherwise, let t be the latest time in (b_u, a_{u+1}) such that the job at the top of H at time t^- is more than z above J_i.

By the definition of t, the stack H is smaller at time t^+ than at time t^-. Hence LAX completed a job at time t since this is the only place in the code of LAX where the stack size can decrease. By the definition of t, the job at the top of H at time t^+ is no more than z above J_i. Hence the depth of J_i at time t^+ is at most $z+1$. The claim then follows by the description of the amortization scheme. ■

Lemma 30 **During each interval I_u, $1 \leq u \leq s$, the job J_i earns at least**

$$\left(\frac{1}{2^{z+2}}\right) \left\lfloor \frac{b_u - a_u}{\frac{2v_i}{\alpha^z}} \right\rfloor$$

credits.

Proof: Each job that is z above J_i is of length at most v_i/α^z by lemma 1. Hence, for each $0 \leq y \leq \left\lfloor \frac{b_u - a_u}{\frac{2v_i}{\alpha^z}} \right\rfloor - 1$, the algorithm LAX must push a job $J_{f(y)}$ that is z above J_i during the period $[a_u + 2y\frac{v_i}{\alpha^z}, a_u + (2y+1)\frac{v_i}{\alpha^z}]$. If $J_{f(y)}$ is completed by SRPT before time Push$(f(y))$, then J_i gains $1/2^{z+2}$ credits from $J_{f(y)}$ in a SRPT-2 transfer at time Push$(f(y))$. Otherwise, suppose that $J_{f(y)}$ is not completed by *SRPT* before time Push$(f(y))$. Since SRPT could opt to run $J_{f(y)}$ at time Push$(f(y))$, SRPT must complete a job during the period $(a_u + 2y\frac{v_i}{\alpha^z}, a_u + (2y+2)\frac{v_i}{\alpha^z}]$, and this job that SRPT completes will contribute $1/2^{z+2}$ credits to J_i in a SRPT-1 transfer. ■

We are now ready to bound the aggregate credits earned by the job J_i.

Lemma 31 **The total credits earned by J_i in the above amortization scheme is at least $\frac{1}{8}$.**

Proof: For $1 \leq s \leq u$, we say that an interval I_u is *long* if $b_u - a_u \geq \frac{4v_i}{\alpha^z}$, otherwise, we say that I_u is *short*. Let σ be the number of short intervals, and ℓ be the aggregate length of the long intervals. Since $\sum_{u=1}^{s} |I_u| \geq \frac{v_i}{2}$, the aggregate length of the short intervals is at most $\frac{v_i}{2} - \ell$, and hence

$$\sigma \geq \frac{\frac{v_i}{2} - \ell}{\frac{4v_i}{\alpha^z}}$$

Then by lemma 29 and lemma 30 we have that the aggregate credits earned by J_i is at least

$$
\begin{aligned}
& \frac{1}{2^{z+2}}\left(s-1+\sum_{u=1}^{s}\left\lfloor\frac{b_u-a_u}{2v_i/\alpha^z}\right\rfloor\right) \\
\geq\ & \frac{1}{2^{z+2}}\left(s-1+\sum_{\text{long } I_u}\left\lfloor\frac{b_u-a_u}{2v_i/\alpha^z}\right\rfloor\right) \\
\geq\ & \frac{1}{2^{z+2}}\left(s-1+\sum_{\text{long } I_u}\left(\frac{b_u-a_u}{2v_i/\alpha^z}-1\right)\right) \\
=\ & \frac{1}{2^{z+2}}\left(\sigma-1+\sum_{\text{long } I_u}\frac{b_u-a_u}{2v_i/\alpha^z}\right) \\
=\ & \frac{1}{2^{z+2}}\left(\sigma-1+\frac{\ell\alpha^z}{2v_i}\right) \\
\geq\ & \frac{1}{2^{z+2}}\left(\frac{\frac{v_i}{2}-\ell}{\frac{4v_i}{\alpha^z}}-1+\frac{\ell\alpha^z}{2v_i}\right) \\
=\ & \frac{1}{2^{z+2}}\left(\frac{\frac{v_i}{2}+\ell}{\frac{4v_i}{\alpha^z}}-1\right) \\
\geq\ & \frac{1}{2^{z+2}}\left(\frac{\frac{v_i}{2}}{\frac{4v_i}{\alpha^z}}-1\right) \\
=\ & \frac{1}{2^{z+2}}\left(\frac{\alpha^z}{8}-1\right) \\
\geq\ & \frac{1}{8}
\end{aligned}
$$

The last inequality follows since $\alpha \geq 16$ and $z \geq 1$. ∎

The following lemma then follows immediately from lemma 27 and lemma 31.

Lemma 32 $8\,|\mathrm{SRPT}(\mathcal{I})| + 8\,|\mathrm{L{\small AX}}(\mathcal{I})| \geq |\mathcal{D}|$.

We are now ready to give the result that is the goal of this subsection.

Lemma 33 $8\,|\mathrm{SRPT}(\mathcal{I})| + 9\,|\mathrm{L{\small AX}}(\mathcal{I})| \geq C$.

Proof: A push in line (1) or line (2) of LAX that increments the size of H from k to $k+1$ is paid for by the first subsequent pop to reduce H to size k. If the pop occurred in line (5) of the code then LAX completed the popped job. If the pop occurred in line (6) of the code then the job popped in line (6) is in $\mathcal{D}$. The result then follows from lemma 32. ∎

3.4 Comparing LAX and Approximate Optimal Solutions

In this section when we speak of the state of LAX at time t, we mean the state of LAX after it has responded completely to all events at time t. It is important to keep this in mind in order to avoid confusion. The main result of this subsection, and the keystone result in this paper, is lemma 37. Lemma 37 essentially shows that during any period of time during which AOPT started executing four different jobs, it must be the case that LAX executed a push in either line (1) or line (2) of its code. Before preceding to lemma 37 we will need a couple of preliminary lemmas.

Lemma 34 Let $[s, t]$ be a period of time during which LAX does not execute a push in either line (1) or line (2) of its code. Then the value of $v_{h(k)}$ at time t is at least as large as the value of $v_{h(k)}$ at time s. Furthermore, if $k > 1$ at time t, then the value of $v_{h(k-1)}$ at time t is at least as large as the value of $v_{h(k-1)}$ at time s.

Proof: We prove the first claim, the proof of the second is similar. Other than lines (1) and (2), the top job of H can only be changed by one of the pops in lines (3), (5) and (6), or by the push in line (4). In the former case, by lemma 1, the value of $v_{h(k)}$ is not lessened. In the latter case the value of $v_{h(k)}$ is not lessened because $v_i > v_{h(k)}$, and $v_j \geq v_i$ since $J_i \in V$ after the pop in line (3). ■

Lemma 35 There does not exist a J_i and a $t \geq r_i$ such that both $J_i \in V$ and $\alpha x_i \leq v_{h(k)}$ at time t.

Proof: If procedure `Fill` was called at time t we get an immediate contradiction. Otherwise, define time s such that the procedure `Fill` was called at time s but was not called at any time in the interval $(s, t]$. Therefore the only way that H can change during the (s, t) is via the push in line (2). Hence, the value of $v_{h(k)}$ at time t is strictly less than the value of $v_{h(k)}$ at time s. Note that $r_i \leq s$, that J_i is viable at time s, and that J_i has not been in H during the interval $[0, s]$. Hence, `Fill` would have added J_i at time s, contradiction. ■

Lemma 36 For all $J_i \in \mathcal{J}$ either $pr_i = r_i$ or LAX popped a job from H at time pr_i.

Lemma 37 Let J_d be the first child of a job J_c, that is a first child of a job J_b, that is a first child of a job J_a, in AOPT. Then during the time interval $[pr_b, pr_d]$, it must be the case that LAX executed line (1) or line (2) of its code.

Proof: Applying lemma 20, and the fact that AOPT is a forest, we get that $pr_b < pr_c < pr_d$. Assume to reach a contradiction that neither line (1) or (2) of LAX was executed during $[pr_b, pr_d]$.

First assume that LAX pushed J_b on H at time pr_b. By our assumption that no pushes happened in line (1) or (2), we can conclude that this push happened in line (4) of LAX.

Throughout the interval $(pr_b, pr_c]$, it is the case that $v_{h(k)} \geq v_b$ by lemma 34. Applying lemma 20 we get that $\alpha x_c \leq v_b$. By the definition of pr_c, it must be the case that J_c was not in H during the interval $[0, pr_b]$. So if $r_c \leq pr_b$ then we get a contradiction to lemma 35 at time pr_b. Otherwise if $r_c > pr_b$ we get a contradiction to lemma 35 at time $r_c \leq pr_c$.

Therefore it must be the case that LAX didn't push J_b at time pr_b. Then by the definition of pr_b, it was the case that $\alpha x_{h(k)} > v_b$ at time pr_b. By lemma 1, $\alpha x_h(k) \leq v_{h(k-1)}$ at time pr_b. Hence, $v_b < v_{h(k-1)}$ at time pr_b. By lemma 20, $\alpha x_c \leq v_b$. Hence, $\alpha x_c < v_{h(k-1)}$ at time pr_b.

Now assume that LAX pushed J_c on H at time pr_c. By our assumption that no pushes happened in line (1) or (2), we can conclude that this push happened in line (4) of LAX. Throughout the interval $(pr_c, pr_d]$, it is the case that $v_{h(k)} \geq v_c$ by lemma 34. Applying lemma 20 we get that $\alpha x_d \leq v_c$. By the definition of pr_d, it must be the case that J_d was not in H during the interval $[0, pr_c]$. So if $r_d \leq pr_c$ then we get a contradiction to lemma 35 at time pr_c. Otherwise if $r_d > pr_c$ we get a contradiction to lemma 35 at time $r_d \leq pr_d$.

Therefore it must be the case that J_c was not pushed on H at time pr_c. Hence, by lemma 36 it must be the case that at time pr_c, either some job must have been popped from H, or J_c was released.

First consider the case that a pop occurred at time pr_c. Recall that in the third paragraph of this proof we concluded that it must be the case that $\alpha x_c < v_{h(k-1)}$ at time pr_b. Hence, $\alpha x_c < v_{h(k)}$ after the first pop at time pr_c by lemma 34 and lemma 1. If this pop happened in line (5) or (6) then J_c, or some other job, will be pushed on H in line (1) of LAX at time pr_c, a contradiction. If this pop happened in line (3) of LAX then J_c, or some job with larger value, must have been pushed on H in line (4) of LAX. Hence, in this case $v_{h(k)} \geq v_c$ at time pr_c.

Now consider the case that $r_c = pr_c$. In this case, either LAX didn't execute the body of the else statement because $v_{h(k)} \geq v_c$, or LAX must have executed the push in line (4) at time pr_c, and since $J_c \in V$ at that time, we again get that $v_{h(k)} \geq v_c$ at time pr_c. As a result we again get that $v_{h(k)} \geq v_c$ at time pr_c.

So under all possible scenarios it is the case that $v_{h(k)} \geq v_c$ at time pr_c. By lemma 34 it must be the case that during the interval $(pr_c, pr_d]$, we have that $v_{h(k)} \geq v_c$.

Applying lemma 20 we get that $\alpha x_d \leq v_c$. By the definition of pr_d, it must be the case that J_d was not in H during the interval $[0, pr_d)$. So if $r_d \leq pr_c$ then we get a contradiction to lemma 35 at time pr_c. Otherwise if $r_d > pr_c$ we get a contradiction to lemma 35 at time $r_d \leq pr_d$. ■

3.5 The Competitive Ratio

In this section we put the previous results together to compute the final competitive ratio of SRPT and LAX together.

Lemma 38 If BOPT is leafy then $|\text{SRPT}(\mathcal{I})| \geq \frac{\beta}{16}|\text{OPT}(\mathcal{J})|$.

Proof: $|\text{BOPT}| \geq \frac{\beta}{8}|\text{OPT}(\mathcal{J})|$ by lemma 19. $|\text{SRPT}(\mathcal{I})|$ is at least half the number of leaves in BOPT by lemma 24. Hence, $|\text{SRPT}(\mathcal{I})| \geq \frac{1}{2}|\text{BOPT}|$ since BOPT is leafy. ∎

Lemma 39 If BOPT is not leafy, $3C + 6\,|\text{SRPT}(\mathcal{I})| \geq |\text{AOPT}|$.

Proof: We use an amortization argument to pay for the jobs completed in AOPT. We initially give 3 credits to each push executed in line (1) or (2) of LAX, and 3 credits to each leaf of AOPT. Let J_a be an arbitrary job in AOPT. We now break the proof into many cases.

If J_a is a leaf in AOPT, then J_a is paid for by it own credits. Otherwise, let J_b be the first child of J_a in AOPT. By lemma 12, J_b begins execution in AOPT at time pr_b.

If J_b is a leaf in AOPT, then J_a is paid for by the credits in J_b. Otherwise, let J_c be the first child of J_b in AOPT. By lemma 12, J_c begins execution in AOPT at time pr_c.

If J_c is a leaf in AOPT, then J_a is paid for by the credits at J_c. Otherwise, let J_d be the first child of J_c in AOPT. By lemma 12, J_d begins execution in AOPT at time pr_d.

Else J_a is paid for by the push that LAX must have executed in line (1) or line (2) during $[pr_b, pr_d]$ by lemma 37.

Note that no push is charged more than three times, and that no leaf is charged more than three times. The result follows since, by lemma 24, $|\text{SRPT}(\mathcal{I})|$ is at least half of the number of leaves in AOPT. ∎

Theorem 40 For all instances $\mathcal{I}$,

$$|\text{OPT}(\mathcal{I})| \leq (\frac{256}{\beta} + \frac{768}{\beta})|\text{SRPT}(\mathcal{I})| + \frac{864}{\beta}|\text{LAX}(\mathcal{I})|$$

Proof: First assume that BOPT is not leafy. Then

$$\begin{array}{rcll}
|\text{OPT}(\mathcal{I})| & \leq & |\text{OPT}(\mathcal{N})| + |\text{OPT}(\mathcal{J})| & \\
& \leq & |\text{OPT}(\mathcal{N})| + \frac{32}{\beta}|\text{AOPT}| & \text{by lemma 20} \\
& \leq & \frac{64}{\beta}|\text{SRPT}(\mathcal{I})| + \frac{32}{\beta}|\text{AOPT}| & \text{by lemma 26} \\
& \leq & \frac{256}{\beta}|\text{SRPT}(\mathcal{I})| + \frac{96}{\beta}C & \text{by lemma 39} \\
& \leq & (\frac{256}{\beta} + \frac{768}{\beta})|\text{SRPT}(\mathcal{I})| + \frac{864}{\beta}|\text{LAX}(\mathcal{I})| & \text{by lemma 33}
\end{array}$$

Now assume that BOPT is leafy. Then

$$\begin{array}{rcll}
|\text{OPT}(\mathcal{I})| & \leq & |\text{OPT}(\mathcal{N})| + |\text{OPT}(\mathcal{J})| & \\
& \leq & |\text{OPT}(\mathcal{N})| + \frac{16}{\beta}|\text{SRPT}(\mathcal{I})| & \text{by lemma 38} \\
& \leq & \frac{80}{\beta}|\text{SRPT}(\mathcal{I})| & \text{by lemma 26}
\end{array}$$

∎

We get the following corollary by setting $\alpha = 24$. Note that if $\alpha = 24$ then $\frac{1}{\beta} \leq 126$.

Corollary 41 If $\alpha = 24$ then for all instances $\mathcal{I}$,

$$|\textsc{Opt}(\mathcal{I})| \leq 129,024 \cdot |\text{SRPT}(\mathcal{I})| + 108,864 \cdot |\textsc{Lax}(\mathcal{I})|$$

References

[1] S. Baruah, J. Harita, and N. Sharma, "On-line scheduling to maximize task completions", *IEEE Real-time Systems Symposium*, 228–237, 1994.

[2] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang, "On the competitiveness of on-line real-time task scheduling", *Journal of Real-Time Systems*, **4**, 124–144, 1992.

[3] A. Borodin, and R. El-Yaniv, *Online Computation and Competitive Analysis*, Cambridge University Press, 1998.

[4] P. Brucker, *Scheduling Algorithms*, Springer-Verlag, 1995.

[5] K. Christian and K. Pruhs, Personal communication.

[6] S. Irani, and A. Karlin, "Online computation", Chapter 13 of *Approximation Algorithms for NP-hard Problems*, ed. D. Hochbaum, PWS Publishing, 1997.

[7] B. Kalyanasundaram and K. Pruhs "Speed is as powerful as clairvoyance", *IEEE Foundations of Computer Science*, 214–223, 1995.

[8] G. Koren and D. Shasha, "MOCA: A multiprocessor on-line competitive algorithm for real-time systems scheduling", *Theoretical Computer Science*, **128**, 75–97, 1994.

[9] R. Lipton, and A. Tomkins, "Online interval scheduling", *ACM-SIAM Symposium on Discrete Algorithms*, 302–311, 1994.

[10] C. Phillips, C. Stein, E. Torng, and J. Wein, "Optimal time-critical scheduling via resource augmentation", *ACM Symposium on Theory of Computation*, 140 – 149, 1997.

[11] J. Sgall, "On-line scheduling - a survey", *On-Line Algorithms: The State of the Art*, eds. A. Fiat and G. Woeginger, Lecture Notes in Computer Science, Springer-Verlag.

[12] G. Woeginger, "On-line scheduling of jobs with fixed start and end time", *Theoretical Computer Science*, **130**, 5–16, 1994..