# Online Scheduling with Partial Job Values: Does Timesharing or Randomization Help?

Francis Y. L. Chin [*] Stanley P. Y. Fung

Department of Computer Science and Information Systems,
The University of Hong Kong, Hong Kong.
{chin, pyfung} @csis.hku.hk

**Abstract**

We study the following online preemptive scheduling problem: given a set of jobs with release times, deadlines, processing times and weights, schedule them so as to maximize the total value obtained. Unlike traditional scheduling problems, partially completed jobs can get partial values proportional to their amounts processed. Recently Chrobak et al. gave improved lower and upper bounds [1.236, 1.8] on the competitive ratio for this problem, the upper bound being achieved by using timesharing to simulate two equal-speed processors. In this paper we (1) give a new algorithm MIXED-$k$ with competitive ratio $1/(1-(\frac{k}{k+1})^k)$ which approaches $e/(e-1) \approx 1.582$ when $k \to \infty$, by using timesharing to simulate $k$ equal-speed processors; (2) give an equivalent but much more practical algorithm MIX, which is $e/(e-1)$-competitive (independent of $k$), by timesharing the processor with different speeds (depending on the job weights), and use its interesting properties to devise an efficient implementation; (3) improve the lower bound to 1.25 by showing an identical lower bound for randomized algorithms; and (4) prove a lower bound of 1.618 on the competitive ratio when timesharing is not allowed, thus answering an open problem raised by Chang and Yap, showing that timesharing provably helps in giving better algorithms for this problem.

**Keywords:** On-line algorithms, scheduling, partial job values, timesharing, lower bounds.

[*]This work is supported in parts by an Earmarked Research Grant (RGC).

## 1 Introduction

Scheduling problems have many different kinds of objective functions, such as minimizing the makespan, completion time, flow time, etc. One of the objective functions is to maximize the total value received for completing the jobs, and partially-processed jobs (i.e., those cannot be completed before the deadline) receive no value. In this paper we consider a variation of the objective function in which partially-processed jobs also get a value proportional to the amount processed. Chang and Yap gave an application in multimedia content transmission over a network with low bandwidth [5]. A related problem known as *imprecise computation* has been studied in real-time systems literature [11], in which a number of different applications were pointed out, including numerical computation, heuristic search, database query processing, etc.

The algorithm needs to schedule the jobs *online*, i.e., the jobs are only known when they arrive and the algorithm cannot make changes to the schedule in the past. Online algorithms are usually analyzed in terms of their competitive ratios, introduced in [13]. An online algorithm is called *c-competitive* if, for any instance of jobs, the value produced by the online algorithm is at least $1/c$ that of the offline optimal algorithm. Competitive analysis and various forms of online scheduling are discussed in detail in [4, 12].

We first define some terms. A *request* (or *job*) is specified by a 4-tuple $(s, d, p, w)$ where $s$ is the *release time*, $d$ is the *deadline*, $p$ is the *processing time* (i.e., length), and $w$ is the *weight* (also called 'value density', i.e., the value received per unit time of processing) of the job. For a job $q$, these parameters are sometimes denoted by $s(q), d(q), p(q)$ and $w(q)$. These are known immediately when the job arrives. The *span* of a job is the time interval $[s(q), d(q)]$. Scheduling is done in a uniprocessor setting, i.e., at most one job can be processed at any time moment. Preemption is allowed at no penalty (jobs are resumed at the point last preempted). A job is *active* if it is released, not yet completed and its deadline has not been reached. A job $q$ processed for a total time $l(q)$ gets a *value* (also called 'profit') $l(q) \times w(q)$.

For the traditional all-or-no-value model, tight bounds are known; the competitive ratio cannot be better than $(1 + \sqrt{B})^2$ [2], and this ratio is achieved by Algorithm $D^{over}$ [10], where $B$ denotes the *importance ratio*, i.e., the ratio of maximum to minimum job weights. For the partial job value model, two heuristics are described in [5]:

- FIRSTFIT: always serves the heaviest job.
- ENDFIT: starting from the heaviest job, allocates each job to the latest possible timeslot(s) within its span.

Both heuristics are shown to be 2-competitive and the bounds are tight. In fact, ENDFIT always gives the offline optimal schedule.

In [7], a 1.8-competitive algorithm MIXED was given for the partial job value model. The algorithm makes use of *timesharing*, i.e., it allows more than one job running on the processor in parallel, each at reduced speeds so that the sum of processing speeds at any time does not exceed the processor speed. This can be simulated by alternating the jobs at a very high frequency. In MIXED, two jobs are running in parallel (each with half the speed): one is the heaviest job, while the other is the earliest deadline job among those active ones whose weights are above a certain threshold. The two may actually be the same job, in which case that job is processed with full speed.

To date no non-timesharing algorithms have a competitive ratio better than 2 if the importance ratio $B$ approaches $\infty$. (To be precise, a $(2\lceil \log B \rceil + 3)/(\lceil \log B \rceil + 2)$-competitive algorithm is given in [6].) Thus it seems the improvement of the 1.8-competitive algorithm stems from timesharing (in fact we show that timesharing does help later in this paper). It is natural to ask why only two jobs are processed simultaneously in MIXED; if timesharing does help, why not try to schedule more jobs concurrently? In Section 2 we generalize the MIXED algorithm to have a competitive ratio $\frac{1}{1-(\frac{k}{k+1})^k}$ for any $k$ where $k$ is the number of equal-speed processors we simulate. This bound is tight, and tends to $e/(e-1)$ as $k$ tends to infinity. Seemingly, the number of jobs running in parallel should be at most the number of currently active jobs; however MIXED-$k$ needs a very large $k$ to be close to $e/(e-1)$-competitive even if the number of jobs is small. Since it is impractical to have $k$ much larger than the number of jobs, the bound might seem only theoretically interesting but unachievable. However in Section 3 we further give an equivalent algorithm MIX that is simple to execute in practice and is always $e/(e-1)$-competitive, independent of $k$ [1]. Intuitively, the timesharing jobs with different weights should not be processed at the same speed; the heavier jobs should get a larger share of the processing power while the lighter jobs get less. This is unlike MIXED in [7] in which if two jobs

[1]The same algorithm is recently independently discovered by Chrobak et al. in the revised version of [7].

are running in parallel, they each run at half the speed, independent of their weights. We will show later that MIX will assign different processing speeds to jobs with different weights (to be precise, each job weight relative to others). Moreover, MIX runs in $O(\min(m, \ell \log m \log(m/\ell)))$ time per re-scheduling, where $m$ is the number of currently active jobs and $\ell$ is the number of changes in the schedule.

A question raised in [5] is whether timesharing helps in reducing the competitive ratio. In Section 4 we prove a lower bound of 1.618 for algorithms that do not use timesharing, thus showing that timesharing provably helps in this problem. Another question raised in [5] is about whether randomization helps in this problem. Although we give no randomized algorithms in this paper, we prove in Section 5 a lower bound of 1.25 on the competitive ratio for any randomized algorithms. This also implies an improved deterministic lower bound for the problem, an improvement of the previously known lower bound result of 1.236.

## 2 An Improved Timesharing Algorithm

### 2.1 The Algorithm

The MIXED algorithm [7] can be thought of using two (virtual) processors, each with half the speed. In this section, we describe and analyze a natural extension of MIXED: instead of two processors, we simulate $k$ equal-speed processors where $k$ is any positive integer.

Our new algorithm MIXED-$k$ is as follows. We simulate $k$ processors $P_1, P_2, ..., P_k$, each getting $1/k$ of the total processor time (i.e., runs at speed $1/k$). At any time, processor $P_i$ runs the job which has the earliest deadline among those that have weights $\geq \alpha_i w_1$, where $w_1$ is the weight of the currently heaviest active job. $\alpha_i$ ($1 = \alpha_1 > \alpha_2 > ... > \alpha_k > 0$) are constants to be specified later. The fact that $\alpha_1 = 1$ means $P_1$ always schedules the heaviest job. Note that it may happen that more than one processor picks the same job, in which case that job runs at a higher speed (e.g. speed $2/k$ if it is picked up by two processors). The schedule remains unchanged until some job is completed, some job reaches its deadline, or a new job arrives.

## 2.2 Analysis

Let $S(t)$ denote the job running by Algorithm $S$ at time $t$. Define $done_S(q,t)$ to be the amount of job $q$ done by time $t$ using Algorithm $S$. Without leading to any confusion, $S$ may stand for the algorithm or the schedule produced by the algorithm interchangably. A schedule $S$ is called *canonical* if, for any two times $t_1$ and $t_2$ ($t_1 < t_2$), the following is satisfied: if $q_1 = S(t_1)$, and $q_2 = S(t_2)$ is not null, then either (i) $s(q_2) > t_1$, or (ii) $q_1$ is not null and $d(q_1) \leq d(q_2)$. Intuitively, it means that among the active jobs at any time, $S$ will either do the one with the earliest deadline [2] , or discard it forever. Let OPT denote the offline optimal schedule. As stated in [7], any schedule (including OPT) can be converted into a canonical schedule. In the proof we will use this property of OPT.

We will use the following *charging scheme* in the proof of competitiveness, which is similar to that in [7]. We charge the values of infinitesimally small time periods (i.e. the 'value rates' or weights) from OPT to those in MIXED-$k$. Let $F : \Re \rightarrow \Re$ be a function mapping each time in OPT to a time in MIXED-$k$. $F$ is defined as follows. For any time $t$, let $q$ be the job running in OPT at time $t$. If $done_{OPT}(q,t) > done_M(q,t)$ (where $M$ denotes our algorithm MIXED-$k$), $F(t) = t$ and the value rate charged is $w(q)$. Otherwise, $F(t) = u$ where $u \leq t$ is the earliest time such that $done_{OPT}(q,t) = done_M(q,u)$ and the value rate charged is $v(q)w(q)$, where $v(q)$ is the speed share of which $q$ is running in MIXED-$k$ at time $u$. (That is, if $i$ out of $k$ processors are running $q$, then $v(q) = i/k$.) See Figure 1. It is easy to see that all values in OPT are charged under mapping $F$. We need to make sure that sum of values charged to any particular time in MIXED-$k$ is not 'too large' (compared to the value it gets). We are going to bound the competitive ratio by bounding the ratio of the charges assigned to MIXED-$k$ to the value attained, for any time $t$.

Suppose at time $t$, $q_0$ is the job running in OPT, and $q_1 ... q_k$ are jobs running at $P_1 ... P_k$. Note that $q_i$'s may not be distinct; in the following the notation $q_u = q_v$ means $q_u$ and $q_v$ are the same job. For clarity define $w_i = w(q_i)$. By definition we have $\alpha_i w_1 \leq w_i \leq w_1$ for $i = 1, 2, ..., k$.

**Lemma 1** *If $F(t) = t$, $q_m$ is the largest-indexed job that charges to time $t$, and $q_0 \neq q_m$, then $w_0 \leq \alpha_m w_1$.*

[2] Assume ties on deadlines are broken consistently for OPT and MIXED-$k$.

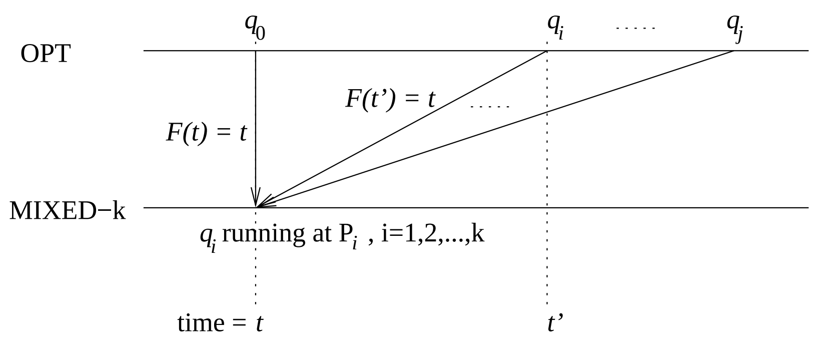


Figure 1: Charging scheme.

*Proof.* Since $F(t) = t$, $q_0$ must be unfinished in MIXED-$k$ at time $t$. $q_m$ must charge from a later time in OPT. Since OPT is canonical, $d(q_0) \leq d(q_m)$, but $P_m$ picks $q_m$ instead of $q_0$, it must be the case that $w_0 \leq \alpha_m w_1$. □

**Lemma 2** *If $F(t) = t$ and $q_0 = q_m$ for some $m > 0$, then $q_m, q_{m+1}, ..., q_k$ cannot charge to time $t$ from a later time in OPT.*

*Proof.* Since $F(t) = t$, $done_{OPT}(q_m, t) > done_M(q_m, t)$, thus $done_{OPT}(q_m, u) > done_M(q_m, t)$ for all $u > t$. Therefore $q_m$ cannot charge to time $t$ from a later time in OPT.

Consider any $q_i$, $m+1 \leq i \leq k$. If $q_i$ charges to $t$ from a later time, then since OPT is canonical, $d(q_0) \leq d(q_i)$, and $w(q_0) = w(q_m) \geq \alpha_m w_1 > \alpha_i w_1$. Thus $P_i$ should pick $q_0$ (or other jobs with earlier deadlines) instead of $q_i$. The only possibility is then $q_i = q_0 (= q_m)$, but then $q_i$ cannot charge to $t$ by the same reason as $q_m$ above. □

**Theorem 1** *MIXED-$k$ is $\dfrac{1}{1 - (\frac{k}{k+1})^k}$-competitive, and the bound is tight.*

*Proof.* Consider the charges to MIXED-$k$ at any time $t$.
Case 1. $F(t) < t$. The competitive ratio is $c \leq \dfrac{\frac{1}{k}(w_1 + ... + w_k)}{\frac{1}{k}(w_1 + ... + w_k)} = 1$.
Case 2. $F(t) = t$. Note that $w_0 \leq w_1$ since $q_0$ must be unfinished ($F(t) = t$) and $P_1$ picks $w_1$.
Case 2.1. $q_0$ is not one of $q_1, ..., q_k$. Suppose $q_m$ is the job with the largest index that charges to $t$, $1 \leq m \leq k$. By Lemma 1, $w_0 \leq \alpha_m w_1$. Thus

$$\begin{aligned} c &\leq \frac{w_0 + \frac{1}{k}(w_m + ... + w_1)}{\frac{1}{k}(w_k + ... + w_1)} \leq \frac{k\alpha_m w_1 + \sum_{i=1}^{m} w_i}{\sum_{i=1}^{k} w_i} = \frac{k\alpha_m w_1 - \sum_{i=m+1}^{k} w_i}{\sum_{i=1}^{k} w_i} + 1 \\ &\leq \frac{k\alpha_m w_1 - \sum_{i=m+1}^{k} \alpha_i w_1}{\sum_{i=1}^{k} \alpha_i w_1} + 1 = \frac{(k+1)\alpha_m + \alpha_{m-1} + ... + \alpha_1}{\sum_{i=1}^{k} \alpha_i} \end{aligned}$$

If there is no such $m$, then we also have $w_0 \leq w_1 = \alpha_1 w_1$, so

$$c \leq \frac{w_0}{\frac{1}{k}(w_k + ... + w_1)} \leq \frac{k\alpha_1 w_1}{\alpha_k w_1 + ... + \alpha_1 w_1} = \frac{k\alpha_1}{\sum_{i=1}^{k} \alpha_i} < \frac{(k+1)\alpha_1}{\sum_{i=1}^{k} \alpha_i}$$

Case 2.2. $q_0 =$ (at least) one of $q_i$. Let $m$ be the smallest $i$ such that $q_0 = q_i$, i.e. $q_0 = q_m$ but $q_0 \neq q_j$ for any $1 \leq j < m$. Then $q_m, q_{m+1}, ..., q_k$ cannot charge to $t$ from a later time (Lemma 2), i.e., only $q_1, ..., q_{m-1}$ may charge to time $t$ from a later time. Let $q_j$ be the one among these jobs that charge to $t$ with the largest index. Then since $q_0 \neq q_j$, $w_0 \leq \alpha_j w_1$ (Lemma 1), and $c \leq \dfrac{w_0 + \frac{1}{k}(w_j + ... + w_1)}{\frac{1}{k}(w_k + ... + w_1)}$ which is the same as Case 2.1. If there is no such $j$, then $w_0 \leq w_1 = \alpha_1 w_1$, which is also the same as Case 2.1.

Therefore in all cases the competitive ratio is bounded by

$$\max\left(\frac{(k+1)\alpha_1}{\sum \alpha_i}, \frac{(k+1)\alpha_2 + \alpha_1}{\sum \alpha_i}, \frac{(k+1)\alpha_3 + \alpha_2 + \alpha_1}{\sum \alpha_i}, ..., \frac{(k+1)\alpha_k + \alpha_{k-1} + ... + \alpha_1}{\sum \alpha_i}\right)$$

By successive substitution and $\alpha_1 = 1$, it could be shown that all the above expressions are equal when $\alpha_i = (\frac{k}{k+1})^{i-1}$. The competitive ratio in this case becomes

$$c \leq \frac{k+1}{\sum_{i=0}^{k-1} (\frac{k}{k+1})^i} = \frac{k+1}{\frac{1-(k/(k+1))^k}{1-k/(k+1)}} = \frac{1}{1-(\frac{k}{k+1})^k}$$

We now show this bound is tight. Consider a set of $k$ instances. For each $1 \leq m \leq k$, the following instance (Fig. 2) consists of $k+1$ jobs (where $\epsilon$ is a very small positive number):

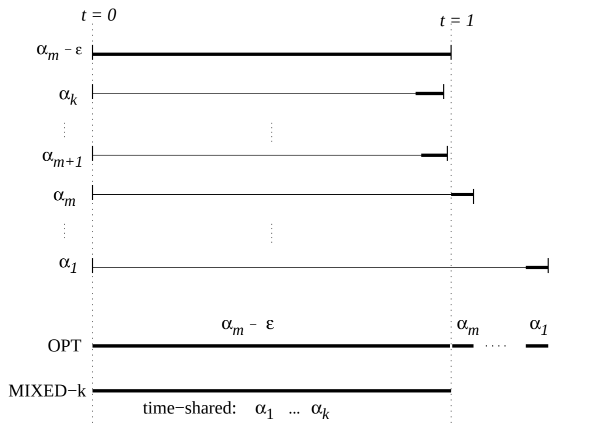
$(0, 1, 1, \alpha_m - \epsilon)$;

for $i = 1, 2, ..., m$, $(0, 1 + (m+1-i)/k, 1/k, \alpha_i)$;

for $i = m+1, ..., k$, $(0, 1 - (i-m)\epsilon, 1/k, \alpha_i)$.

OPT schedules the long $\alpha_m - \epsilon$ job in time [0,1] and then the short jobs $\alpha_m, ..., \alpha_1$ in the remaining time. MIXED-$k$ schedules the short $\alpha_1, ..., \alpha_k$ jobs in time [0,1], just completing all of them at $t = 1$ (since each has length $1/k$ and speed $1/k$). Thus each instance corresponds to a competitive ratio of $\frac{(k+1)\alpha_m + \alpha_{m-1} + ... + \alpha_1}{\sum \alpha_i}$. Therefore the bounds are achievable. In fact the choice of $\alpha_i$'s is the best possible: setting $\alpha_i$'s as in the algorithm gives $\frac{1}{1-(\frac{k}{k+1})^k}$ for all expressions, and changing any $\alpha_i$ will increase the value of at least one of the expressions. □

**Corollary 1** *Algorithms FIRSTFIT [5] and MIXED [7] are special cases of MIXED-k with $k = 1$ and 2, and with competitive ratios 2 and 1.8 respectively. Theorem 1 also gives a correct competitive ratio for them.*



Horizontal lines represent job spans; thick portions represent job lengths.
Bottom two lines are the optimal and online schedules respectively.

Figure 2: Tight examples for MIXED-$k$.

**Corollary 2** *When $k \to \infty$, the competitive ratio of MIXED-k tends to* $e/(e-1) \approx 1.582$.

*Proof.* Immediate from the Theorem 1 and the fact that $\lim_{k\to\infty}(1-1/k)^k = 1/e$. □

## 3 An $e/(e-1)$-competitive Algorithm

To achieve a competitive ratio of $e/(e-1)$ by MIXED-$k$, we need to simulate an infinite number of processors, which of course is not feasible. However, note that among the infinite number of processors, at most $m$ distinct jobs are being scheduled, where $m$ is the number of currently active jobs. They occupy different fractions of the infinite number of (virtual) processors, i.e., getting different proportions of (real) processor time, or equivalently, running at different speeds. We are not interested in which (virtual) processor picks up which job; we only want to know the 'speed share' (i.e. portion of virtual processors) obtained by each job. In this section we discuss how to transform the above algorithm into a practical one when $k = \infty$ and to calculate efficiently the speed each job can share in each re-scheduling when some job is completed, reaches its deadline or is newly released.

### 3.1 The Algorithm

At any time the algorithm maintains a list of active jobs $q_1, q_2, ..., q_m$ with weights $w_1, w_2, ..., w_m$. For simplicity assume $1 = w_1 \geq w_2 \geq ... \geq w_m$. The list is always sorted in decreasing order of job weights. Note that $w_1$ is always 1 through normalization even though $w(q_1)$ may be different at different times.

As $\alpha_k \to 1/e$ when $k \to \infty$, jobs lighter than $1/e \approx 0.37$ of the currently heaviest job will not get any processor time. Since we have an infinite number of processors, we have an infinite number of $\alpha_i$ values covering the range $(1/e, 1]$. Different $\alpha_i$ values define the sets of jobs that can be selected by the respective processors ($P_i$ can only choose among jobs with weights $\geq \alpha_i w_1$). Note that the set of jobs grows as $\alpha_i$ decreases. Since there are $m$ active jobs, there are at most $m$ different such sets. Denote them by $S_i = \{q_i, q_{i-1}, ..., q_2, q_1\}, 1 \leq i \leq m$. The infinite number of virtual processors are therefore partitioned into $m$ (unequal-sized) groups. Processors in each group, although having different $\alpha_i$'s, pick the earliest-deadline job among the same set of jobs, and hence will pick the same earliest-deadline job. We are going to compute the proportion of processors got by each such set.

Consider the job set $S_i$. Let $q_{ED(i)}$ be the earliest-deadline job in $S_i$. We want to find the proportion of processors that choose $q_{ED(i)}$. Since we have infinitely many $\alpha_j$'s, we must have $\alpha_j \approx w_{i+1}$ for some $j$. Thus

$$\alpha_j = \left(\frac{k}{k+1}\right)^{j-1} = w_{i+1}$$

Solving for $j$ gives

$$j = \frac{\ln w_{i+1}}{\ln(k/(k+1))} + 1$$

Since $\lim_{k\to\infty} \frac{1/k}{\ln(k/(k+1))} = -1$,

$$\lim_{k\to\infty} \frac{j}{k} = (\ln w_{i+1}) \lim_{k\to\infty} \frac{1/k}{\ln(k/(k+1))} = -\ln w_{i+1}$$

This means a fraction $-\ln w_{i+1}$ of processors have $\alpha_j > w_{i+1}$. Similarly a fraction of $-\ln w_i$ of processors have $\alpha_j > w_i$. Thus the proportion of processors picking earliest-deadline job from $S_i$ (i.e., it picks $q_{ED(i)}$) is $-\ln w_{i+1} - (-\ln w_i) = \ln(w_i/w_{i+1})$.

The above is true if $w_i > w_{i+1} > 1/e$. If $w_{i+1} < 1/e$, or $q_i$ is the lightest job, the above calculation does not work. In this case $q_{ED(i)}$ takes all the remaining portion of processors. Jobs lighter than $1/e$ get no portion of processors. The general case thus can be summarized by the formula

$$\text{speed share of } q_{ED(i)} \quad = \quad \ln\left(\frac{\max(w_i, 1/e)}{\max(w_{i+1}, 1/e)}\right) \qquad (1)$$

for $1 \leq i \leq m$ (with a dummy $w_{m+1} = 0$).

Now instead of simulating an infinite number of processors, we are reduced to three tasks. First, find $q_{ED(i)}$ for each set $S_i$. Next, for each of the $m$ jobs (possibly non-distinct) found, compute the portion of processor time it occupies using (1). The final task is to find out if any $q_{ED(i)}$'s are identical, in which case we need to sum up the speed shares allocated. The algorithm is presented below.

```
Algorithm MIX
/* a list of currently active jobs $q_1, ..., q_m$ is maintained in decreasing order of weights
(ties broken arbitrarily) at all times */
If a new job arrives, or if a job is finished (either completed or reached its deadline):
    Step 1: insert it into/delete it from the list
    Step 2: compute $q_{ED(i)}$ = earliest-deadline job among $S_i$ for all $i$
    Step 3: compute speed shares of each $q_{ED(i)}$ using (1)
    Step 4: add up speed shares of duplicates
```

As an example, consider the instance of three jobs $q_1(0, 1, 1, 1-\epsilon)$, $q_2(0, 1, 1/2, 2/3)$, $q_3(0, 3/2, 1/2, 1)$ where $\epsilon$ is very small (this is a worst-case example of the MIXED algorithm in [7]). At $t = 0$, MIX will assign $\ln(3/2) \approx 40.5\%$ of processing speed to $q_1$ and the remaining 59.5% to $q_2$, and $q_3$ receives virtually no speed as $\ln(1/(1-\epsilon)) \approx 0$ (this is reasonable since its weight is almost the same as $q_1$ but has a later deadline.) However MIXED in [7] will assign 50% speed to both $q_2$ and $q_3$.

## 3.2 Time Complexity

Algorithm MIX takes $O(m)$ time per re-scheduling. For Step 1, the sorted list of jobs can be easily maintained in $O(m)$ time. For Step 2, $q_{ED(i)}$ can be computed incrementally since $q_{ED(i)}$ = earlier-deadline job among $\{q_i, q_{ED(i-1)}\}$. Thus it takes $O(m)$ time in total. Step 3 clearly takes $O(m)$ time. Step 4 can also be done in $O(m)$ time because the $q_{ED(i)}$'s are already in sorted order of weights; we only need to check for duplicates in adjacent items.

The above linear time bound is worst-case optimal as there are cases which the schedule is completely changed for each re-scheduling. For example, if the heaviest job is also the earliest deadline job, then it occupies all the processor time, and when it is deleted (finished) the updated schedule needs to be recomputed and may have $\Theta(m)$ jobs running at different speeds. However, the algorithm has very interesting properties which can be utilized to give better time complexity in some cases. For example, the job speeds are determined by the weight *ratios* between jobs. Suppose $q_{ED(i)} = q_i$ for $i$=1,2,3, $q_{ED(1)}$ gets speed $\ln(w_1/w_2)$, $q_{ED(2)}$ gets speed $\ln(w_2/w_3)$, and now $q_2$ is deleted. Then new speed of $q_{ED(1)} = \ln(w_1/w_3) = \ln(w_1/w_2) + \ln(w_2/w_3)$, i.e., the speed for $q_{ED(2)}$ is 'reallocated' to $q_{ED(1)}$ without affecting other job speeds, so it seems that the update can be done in constant time in this case. Moreover, Step 2 is essentially computing a *prefix minimum* on the deadlines, i.e. $\min_{w_j \geq w_i}\{d(j)\}$ for all $i$, and we have the following lemma with its proof in the Appendix.

**Lemma 3** *Given a set of jobs* $\{q_1, ..., q_m\}$ *where each job* $q_i$ *is associated with a weight* $w_i$ *and a deadline* $d_i$*, there is a data structure that supports inserting and deleting a job (Step 1) and computing the prefix minimum* $\min_{w_j \geq w_i}\{d_j\}$ *for any* $1 \leq i \leq m$ *(Step 2) in* $O(\log m)$ *time.*

Let us now revisit the problem of computing the updated schedule. In the updated schedule, it may happen that many $q_{ED(i)}$'s are identical; we want to avoid computing all these redundancies. If we plot the jobs in a graph of deadlines against weights (Fig. 3), the 'staircase' lines reveal the distinct $q_{ED(i)}$'s. We want to compute this staircase (that is, the 'dominant jobs' in the staircase, where a job is *dominant* if no other job has both heavier weight and earlier deadline than it). Once this is done we can immediately know the distinct $q_{ED(i)}$'s and their speed shares in $O(L)$ time (analogous to formula (1)), where $L$ is the length of the output, i.e., number of jobs getting nonzero speeds in the updated schedule.

One obvious approach for computing the staircase is sequential search: compute the prefix minimum of the $i$ heaviest jobs for all $i$. This takes $O(m)$ queries. (By 'a query' we mean using the data structure to find a particular prefix minimum.) A better method is to use binary search: starting from the heaviest job, find the next heaviest job that has a smaller prefix minimum deadline by binary search, using $O(\log m)$ queries. Then find the second one, etc. This takes $O(L \log m)$ queries and $O(L \log^2 m)$ time by Lemma 3. The following lemma (proof in Appendix) shows a better method to compute the staircase using the *unbounded search* technique [3].

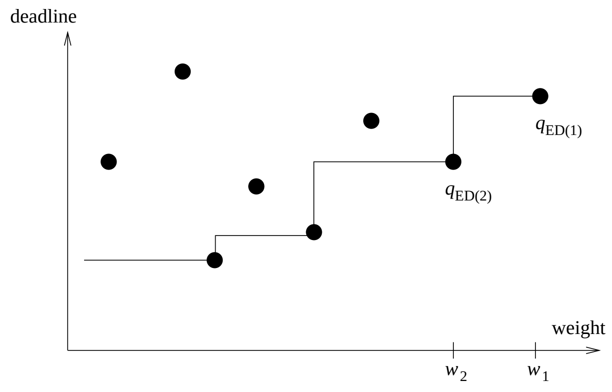


Figure 3: The 'staircase'.

**Lemma 4** *The $L$ jobs getting nonzero speeds (the $L$ dominant jobs of the staircase) can be found in $O(L \log m \log(m/L))$ time.*

This bound is better than the $O(m)$ bound when $L$ is small (e.g., $O(\log^2 m)$ when $L$ is constant), but at worst it gives $O(m \log m)$. We can use a simple trick to guarantee a worst case $O(m)$ time. While processing as above we count $L$, and when we discover that $L > m/\log^2 m$, we abandon the processing and use the simple $O(m)$-time MIX algorithm from scratch instead. (Note that when this happens, we need to retrieve all jobs in sorted order in $O(m)$ time. This can be done by threading all leaves of the tree with a linked list.)

Note that jobs lighter than $1/e$ get no speed and thus should not be outputted. Thus in the algorithm, when we encounter a job with weight $< 1/e$ for the first time, we should compute the speed of the corresponding $q_{ED(i)}$ with $1/e$ replacing that small weight, and then stop the algorithm.

In fact, we can use the above method to compute just the updated part of the schedule. The staircase only changes from the point of insertion/ deletion towards the left until it meets the original staircase where we can stop. (The insertion or deletion of the heaviest job changes the jobs cut off by the '$1/e$ bound' mentioned above, and they need to be updated also.) With some careful handling, we can keep the running time with $L$ replaced by $\ell$, the number of changes in the schedule.

**Theorem 2** *Computing the new schedule can be done in $O(\min(m, \ell \log m \log(m/\ell)))$ time per re-scheduling, where $m$ is the number of currently active jobs and $\ell$ is the number of changes in* *the schedule.*

## 4 A Lower Bound for Non-timesharing Algorithms

In practice, timesharing can be achieved by switching the jobs at a very high frequency. This is a major drawback since it is costly. We might disallow timesharing by specifying that an algorithm cannot change its job at infinitesimally small time intervals. However, most (non-timesharing) heuristics may change their job at *event points* (i.e., times with jobs released/completed/reached their deadlines), and two event points may be very close to each other (depending on the job parameters). Some heuristics like ENDFIT may even change its job at times other than event points. To cater for this, let $X = \{x_i\}$ contains all the time parameters of all 'residue' jobs (the part of the unprocessed job) released so far. Assuming current time $= 0$, then $X$ contains the relative deadlines and remaining processing times of the jobs. For a *non-timesharing* algorithm, the earliest time it can change its job is when new job(s) arrive, or the time that can be formed by adding/subtracting numbers in $X$; in other words, all times of the form $\Sigma a_i x_i$ where $a_i \in \{-1, 0, 1\}$. In particular, if all time parameters of jobs are integers, then non-timesharing algorithms can only change its job at integral times. It is not difficult to see that FIRSTFIT and ENDFIT are non-timesharing under this definition.

We show that in the non-timesharing case there is a stronger lower bound than the currently best bound for the case with timesharing allowed (it is shown that the bound is 1.25 in Section 5).

**Theorem 3** *No deterministic non-timesharing algorithm can have competitive ratio better than $\phi$, where $\phi = (\sqrt{5}+1)/2 \approx 1.618$ is the golden ratio.*

*Proof.* The proof modifies the technique used in [7]. We show that no online algorithms can have competitive ratio $\phi - \epsilon$ for any $\epsilon > 0$. Let $\sigma = \sqrt{5} - 2$. Define a sequence $\{v_i\} : v_0 = 1, v_1 = \phi + \epsilon$, and $v_{i+1} = (v_i - v_{i-1})/\sigma$ for $i > 1$. Solving the recurrence gives $v_i = (1-\epsilon)\phi^i + \epsilon(\phi+1)^i$ for $i \geq 0$. Note that $\lim_{k\to\infty}(v_k/v_{k-1}) = \phi + 1$, and $\sum_{i=2}^{n} v_i = (v_{n-1} - v_0)/\sigma$. These facts will be needed later.

In the following the adversary will give jobs with all time parameters being integers. Thus with our non-timesharing definition above, any non-timesharing algorithm can only schedule

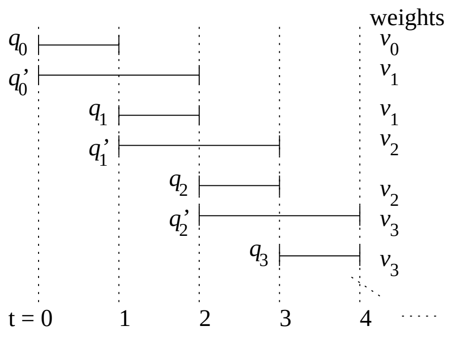


Figure 4: Lower bound construction.

one job at any integral timeslot.

At each integer time $i = 0, 1, 2, ...$, the adversary releases two jobs $q_i(i, i+1, 1, v_i), q_i'(i, i+2, 1, v_{i+1})$ (see Fig. 4). Let $k$ be a sufficiently large integer. If there is an integer time $1 \leq j < k$ such that the algorithm does $q_{j-1}'$ in the slot finishing in time $j$, the adversary stops releasing further jobs. In this case optimal value $= v_1 + ... + v_{j-2} + 2v_{j-1} + v_j$, while the online algorithm gets at most $v_0 + v_1 + ... + v_{j-2} + v_j$. Thus the competitive ratio

$$\begin{aligned}
c &\geq \frac{(v_1+v_2+...+v_j)+v_{j-1}}{(v_0+v_1+...+v_j)-v_{j-1}} = 1 + \frac{2v_{j-1}-1}{(v_0+v_1+...+v_j)-v_{j-1}} = 1 + \frac{2v_{j-1}-1}{v_0+v_1+\frac{v_{j-1}-1}{\sigma}-v_{j-1}} \\
&= 1 + \frac{\sigma(2v_{j-1}-1)}{\sigma+\sigma v_1+(1-\sigma)v_{j-1}-1} = 1 + \frac{2\sigma(v_{j-1}-1/2)}{(1-\sigma)(v_{j-1}-1/2)+\sigma v_1+(\sigma-1)/2} \\
&= 1 + \frac{2\sigma(v_{j-1}-1/2)}{(1-\sigma)(v_{j-1}-1/2)+\sigma\epsilon} \geq 1 + \frac{2\sigma}{1-\sigma+2\sigma\epsilon} = 1 + \frac{2\sigma}{1-\sigma}\left(\frac{1}{1+\frac{2\sigma\epsilon}{1-\sigma}}\right) \\
&> 1 + \frac{2\sigma}{1-\sigma}\left(1-\frac{2\sigma\epsilon}{1-\sigma}\right) > 1 + \frac{2\sigma}{1-\sigma} - \left(\frac{2\sigma}{1-\sigma}\right)^2\epsilon > \phi - \epsilon
\end{aligned}$$

Otherwise, the adversary releases all jobs up to time $k-1$, and at time $k$ releases job $q_k$ only. In this case optimal value $= (v_1 + ... + v_k) + v_k$, while the online algorithm gets at most $v_0 + v_1 + ... + v_k$. Thus the competitive ratio

$$c \geq \frac{v_1+...+v_k+v_k}{v_0+v_1+...+v_k} = 1 + \frac{v_k-v_0}{v_0+v_1+...+v_k} = 1 + \frac{v_k-1}{1+v_1+(v_{k-1}-1)/\sigma} \to 1 + \frac{\phi+1}{1/\sigma} = \phi$$

Thus no algorithms can be better than $\phi$-competitive. □

The competitive ratio 1.58 achieved by MIX is lower than the 1.618 lower bound for the non-timesharing case. This shows that (in this partial job value model) timesharing is provably better than non-timesharing, and their competitive ratios can be significantly different.

## 5 Lower Bound for Randomized Algorithms

In [5] the deterministic lower bound of $2(2-\sqrt{2}) \approx 1.17$ on the competitive ratio was given for this problem, and Chrobak et al. improved it to $\sqrt{5}-1 \approx 1.236$ [7]. In this section we prove a randomized lower bound for this problem which is better than the above bounds, thus improving the deterministic lower bound at the same time.

We make use of Yao's principle [14]. Basically, it enables us to find a lower bound of randomized algorithms by finding a probability distribution of instances, such that we can bound the ratio of the expected offline optimal value to the expected online value of the best *deterministic* algorithm. This ratio will then be a lower bound of randomized algorithms (see [4]).

**Theorem 4** *No randomized (and hence deterministic) algorithms can be better than 1.25-competitive.*

*Proof.* Consider a set of $n+1$ instances:

$J_1 = \{(0,1,1,1),(0,2,1,2)\}$

$J_i = J_{i-1} \cup \{(i-1,i,1,2^{i-1}),(i-1,i+1,1,2^i)\}$ for $i = 2,...,n$

$J_{n+1} = J_n \cup \{(n,n+1,1,2^n)\}$

The instances are like those of Figure 4 (although the weights are different): $J_1$ corresponds to $\{q_0,q_0'\}$, $J_2$ corresponds to $\{q_0,q_0',q_1,q_1'\}$, etc. We form a probability distribution of $J_i$'s with $p_i$ being the probability of picking $J_i$: $p_i = 1/2^i$ for $i = 1,2,...,n$ and $p_{n+1} = 1/2^n$. Clearly $\sum p_i = 1$.

First consider the offline optimal value. It is easy to see that $OPT(J_i) = (2+2^2...+2^i)+2^{i-1}$ for $i = 1,2,...,n$, and $OPT(J_{n+1}) = 2+2^2...+2^n+2^n$. (Here we overload the symbols $OPT$ and $ALG$ to denote values obtained instead of the schedules.) Thus

$$\begin{aligned}
\mathbf{E}[OPT] &= \sum_{i=1}^{n} \frac{1}{2^i}[(2+...+2^i)+2^{i-1}] + \frac{1}{2^n}[2+...+2^n+2^n] \\
&= \sum_{i=1}^{n} \frac{2(2^i-1)+2^{i-1}}{2^i} + \frac{2(2^n-1)+2^n}{2^n} \\
&= \sum_{i=1}^{n} (\frac{5}{2}-\frac{2}{2^i}) + \frac{3(2^n)-2}{2^n} \\
&= \frac{5n}{2}+1
\end{aligned}$$

Next we consider the online algorithm. At any time interval $[i-1, i]$ where $i$ is an integer, any deterministic algorithm is faced with a heavier job and one or two lighter jobs. Suppose it spends $\beta_i$ units of time in the lighter job(s) (and hence $1-\beta_i$ for the heavier job). The $\beta_i$'s completely determine the value obtained by this algorithm (on these instances). Thus any deterministic online algorithm $ALG$ is specified by a set of numbers $\beta_1, ..., \beta_n$ , $0 \le \beta_i \le 1$.

$$\begin{aligned}
ALG(J_1) &= [\beta_1 + 2(1-\beta_1)] + 2\beta_1 \\
ALG(J_2) &= [\beta_1 + 2(1-\beta_1)] + [2\beta_2 + 4(1-\beta_2)] + 4\beta_2 \\
&\phantom{=} \; ... \\
ALG(J_n) &= [\beta_1 + 2(1-\beta_1)] + ... + [2^{k-1}\beta_k + 2^k(1-\beta_k)] + ... + [2^{n-1}\beta_n + 2^n(1-\beta_n)] + 2^n\beta_n \\
ALG(J_{n+1}) &= [\beta_1 + 2(1-\beta_1)] + ... + [2^{k-1}\beta_k + 2^k(1-\beta_k)] + ... + [2^{n-1}\beta_n + 2^n(1-\beta_n)] + 2^n \\
\mathbf{E}[ALG] &= \frac{1}{2}ALG(J_1) + \frac{1}{4}ALG(J_2) + ... + \frac{1}{2^n}ALG(J_n) + \frac{1}{2^n}ALG(J_{n+1})
\end{aligned}$$

Rearranging $\mathbf{E}[ALG]$ by grouping its constant terms, coefficients of $\beta_1$, etc:

Coefficient of $\beta_1 = \frac{1}{2}(1) + \frac{1}{4}(-1) + \frac{1}{8}(-1) + ... + \frac{1}{2^n}(-1) + \frac{1}{2^n}(-1) = \frac{1}{2} - (\frac{1}{4} + \frac{1}{8} + ... + \frac{1}{2^n} + \frac{1}{2^n}) = 0$

In general, for $1 \le i \le n-1$,

Coefficient of $\beta_i = \frac{1}{2^i}(2^{i-1}) + \frac{1}{2^{i+1}}(2^{i-1} - 2^i) + ... + \frac{1}{2^n}(2^{i-1} - 2^i) + \frac{1}{2^n}(2^{i-1} - 2^i) = 0$, and

Coefficient of $\beta_n = \frac{1}{2^n}(2^{n-1}) + \frac{1}{2^n}(2^{n-1} - 2^n) = 0$

This shows a very interesting fact: the expected online value does not depend on which algorithm is used. $\mathbf{E}[ALG]$ now only depends on the constant terms:

$$\begin{aligned}
\mathbf{E}[ALG] &= \frac{1}{2}(2) + \frac{1}{4}(2+4) + \frac{1}{8}(2+4+8) + ... + \frac{1}{2^n}(2 + ... + 2^n) + \frac{1}{2^n}(2 + ... + 2^n + 2^n) \\
&= \sum_{i=1}^{n} \frac{1}{2^i}(2 + ... + 2^i) + \frac{1}{2^n}(2 + ... + 2^n + 2^n) \\
&= \sum_{i=1}^{n} \frac{2(2^i - 1)}{2^i} + \frac{2(2^n - 1) + 2^n}{2^n} \\
&= \sum_{i=1}^{n} (2 - \frac{2}{2^i}) + 3 - \frac{2}{2^n} \\
&= 2n + 1
\end{aligned}$$

Hence $\dfrac{\mathbf{E}[OPT]}{\mathbf{E}[ALG]} = \dfrac{5n/2 + 1}{2n + 1}$, and is arbitrarily close to $5/4$ as $n$ is very large. Thus no randomized algorithms have competitive ratio better than 1.25. $\square$

## 6 Concluding Remarks

In this paper we improved the upper and lower bounds on the competitive ratio of the partial job value online scheduling problem. It is interesting that the $e/(e-1)$ bound appears in many online problems (see e.g. [9]). We also showed that the ability of timesharing affects the competitive ratio. It remains open what is the true bound of this problem. Is MIX the best possible, or in other words, can we prove a lower bound of $e/(e-1)$ on the competitive ratio? It is also not known whether there are non-timesharing algorithms beating the competitive ratio of 2.

It is also asked in [5] whether randomized algorithms can give better competitiveness. Recently it was pointed out [7] that randomization will not help for timesharing algorithms since any randomized algorithm can be transformed into a deterministic but timesharing one: at any time, each job is processed with speed equal to its probability of being scheduled by the randomized algorithm. Whether randomization helps in the non-timesharing case is however still unknown.

**Acknowledgements.** We would like to thank the anonymous referees for their helpful comments.

## References

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[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, *Real-Time Systems* 4, 125–144, 1992.

[3] J. L. Bentley and A. C.-C. Yao, An Almost Optimal Algorithm for Unbounded Searching, *Information Processing Letters* 5(3), 82–87, 1976.

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

[5] E. Chang and C. Yap, Competitive Online Scheduling with Level of Service, *Proceedings of 7th Annual International Computing and Combinatorics Conference*, 453–462, 2001.

[6] F. Y. L. Chin and S. P. Y. Fung, Online Scheduling with Partial Job Values and Bounded Importance Ratio, *Proceedings of International Computer Symposium*, 787–794, 2002.

[7] M. Chrobak, L. Epstein, J. Noga, J. Sgall, R. van Stee, T. Tichý and N. Vakhania, Preemptive Scheduling in Overloaded Systems, preliminary version appeared in *Proceedings of 29th International Colloqium on Automata, Languages and Programming*, 800–811, 2002.

[8] C.-T. Ho, R. Agrawal, N. Megiddo and R. Srikant, Range Queries in OLAP Data Cubes, *Proceedings of ACM SIGMOD Conference on the Management of Data*, 73–88, 1997.

[9] A. Karlin, C. Kenyon and D. Randall, Dynamic TCP Acknowledgement and Other Stories about $e/(e-1)$, *Proceedings of the Symposium on the Theory of Computing*, 502–509, 2001.

[10] G. Koren and D. Shasha, $D^{over}$: An Optimal On-line Scheduling Algorithm for Overloaded Uniprocessor Real-time Systems, *SIAM Journal on Computing* 24, 318–339, 1995.

[11] J. W. S. Liu, K.-J. Lin, W.-K. Shih, A. C.-S. Yu, J.-Y. Chung and W. Zhao, Algorithms for Scheduling Imprecise Computations, *IEEE Computer* 24(5), 58–68, 1991.

[12] J. Sgall, Online Scheduling, in *Online Algorithms: the State of the Art* (Fiat and Woeginger eds.), Springer-Verlag, 196–227, 1998.

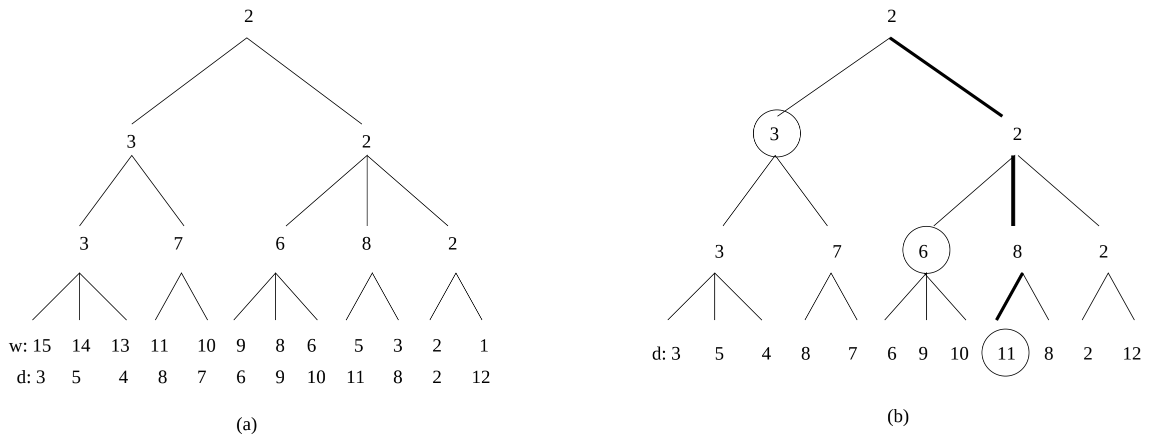
[13] D. Sleator and R. Tarjan, Amortized Efficiency of List Update and Paging Rules, *Communications of the ACM* 28(2), 202–208, 1985.

[14] A. C.-C. Yao, Probabilistic Computations: Toward a Unified Measure of Complexity, *Proceedings of 18th IEEE Symposium on Foundations of Computer Science*, 222–227, 1977.

## A Appendix: Proofs of Lemmas

**Lemma 3** *Given a set of jobs* $\{q_1, ..., q_m\}$ *where each job* $q_i$ *is associated with a weight* $w_i$ *and a deadline* $d_i$*, there is a data structure that supports inserting and deleting a job (Step 1) and computing the prefix minimum* $\min_{w_j \geq w_i}\{d_j\}$ *for any* $1 \leq i \leq m$ *(Step 2) in* $O(\log m)$ *time.*

*Proof.* We use a 2-3 tree [1] to store the jobs according to non-increasing order of job weights. Insertions and deletions can be done in $O(\log m)$ time. We augment prefix-min deadline information on the tree. This technique has been used in range queries in databases (see e.g. [8]). Specifically, in every node we have a field P: for non-leaf nodes it stores the minimum of the P's of its children, and for leaves it stores the deadline of the corresponding jobs. Fig. 5(a) shows such a tree. We also need to update these P values when we insert/delete jobs, but it is easy to see that this can be done in $O(\log m)$ time.



2
3
2
3
7
6
8
2
w: 15 14 13 11 10 9 8 6 5 3 2 1
d: 3 5 4 8 7 6 9 10 11 8 2 12
(a)
2
3
2
3
7
6
8
2
d: 3 5 4 8 7 6 9 10 11 8 2 12
(b)

Figure 5: (a) A 2-3 tree storing 12 jobs, augmented with P values. (b) Finding min-deadline among the heaviest 9 jobs. Thick edges represent edges traversed and the result is obtained by taking the minimum of all circled P values.

Using the P values, we can, given any $i$, find the earliest-deadline job among the $i$ heaviest jobs in $O(\log m)$ time. First, search the tree to locate the leaf corresponding to the $i$-th heaviest job in $O(\log m)$ time. (Each node needs to be augmented with a field indicating the size of the subtree rooted at it to facilitate this search.) Then along the path $v_0 \to v_1 \to v_2$ ..., the prefix minimum can be found by finding the minimum of all the P values of the left siblings of $v_i$ for all $i$, and also that of the leaf being reached. Fig. 5(b) illustrates this process. $\square$

**Lemma 4** *The $L$ jobs getting nonzero speeds (the $L$ dominant jobs of the staircase) can be found in $O(L \log m \log(m/L))$ time.*

*Proof.* We only need 'Algorithm $B_1$' in [3]. Suppose we have an array with elements sorted in non-increasing order (but may have many identical elements), the current (first) element is $x$, and we want to find the first (foremost) element smaller than $x$. We 'probe' the 2nd, 4th, 8th, ... element, doubling the step size each time until we encounter an element (say, the $2^i$-th element) smaller than $x$. Then we use binary search in the interval between the $2^{i-1}$-th and $2^i$-th elements, to find the foremost element smaller than $x$. Suppose it is the $x_1$-th element. Then the forward doubling search reaches at most the $2x_1$-th element, using $\lceil \log(2x_1) \rceil$ probes. Backward binary search adds another $\lceil \log(2x_1) \rceil$. So this takes at most $2\lceil \log(2x_1) \rceil$ probes.

In our problem, we probe into a search tree with prefix-minimum, not a sorted array, and we want to find the dominant jobs of the staircase. Let $x_1, x_2, ..., x_L$ denote the distances (number of jobs) from the previous dominant job, $x_1 + x_2 + ... + x_L = m$. Probing for the $i$-th dominant job takes at most $2\lceil \log(2x_i) \rceil$ queries, thus the total number of queries is at most $\sum_{i=1}^{L} 2\lceil \log(2x_i) \rceil$. This is maximum when all $x_i$'s are equal, thus the number of probes is at most $2L\lceil \log(2m/L) \rceil$. Thus we need $O(L \log(m/L))$ queries to the tree, taking a total of $O(L \log m \log(m/L))$ time.

□