Meeting deadlines: How much speed suffices?*

S. Anand[1], Naveen Garg[1], and Nicole Megow[2]

[1] Indian Institute of Technology Delhi, India.
anand.42@gmail.com, naveen@cse.iitd.ac.in
[2] Max-Planck-Institut für Informatik, Saarbrücken, Germany.
nmegow@mpi-inf.mpg.de

Abstract. We consider the online problem of scheduling real-time jobs with hard deadlines on m parallel machines. Each job has a processing time and a deadline, and the objective is to schedule jobs so that they complete before their deadline. It is known that even when the instance is feasible it may not be possible to meet all deadlines when jobs arrive online over time. We therefore consider the setting when the algorithm has available machines with speed $s > 1$.
We present a new online algorithm that finds a feasible schedule on machines of speed $e/(e-1) \approx 1.58$ for any instance that is feasible on unit speed machines. This improves on the previously best known result which requires a speed of $2 - 2/(m+1)$. Our algorithm only uses the relative order of job deadlines and is oblivious of the actual deadline values. It was shown earlier that the minimum speed required for such algorithms is $e/(e-1)$, and thus, our analysis is tight. We also show that our new algorithm outperforms two other well-known algorithms by giving the first lower bounds on their minimum speed requirement.

1 Introduction

We consider the problem of scheduling real-time jobs with hard deadlines on multiple machines. In this problem, a set of jobs $J = \{1, \ldots, n\}$ must be scheduled on m identical parallel machines, each of which can process at most one job at the time. A job $j \in J$ arrives at its release date $r_j \in \mathbb{N}$, has processing time $p_j \in \mathbb{N}$, and deadline $d_j \in \mathbb{N}$. It can be processed on any of the m machines. It can also be preempted and restarted, either on the same machine or a different machine. An instance is called *feasible*, if there exists a schedule such that no job misses its deadline. An algorithm is *optimal* if it can schedule every feasible instance so that all deadlines are met. Given a feasible instance, a feasible schedule can be computed by solving a maximum flow problem [4].

In this paper, we consider the online model, in which an algorithm learns about a job j only at its release date r_j. Several algorithms are known to be optimal on a single machine [3]. But on multiple machines, the online problem is much more difficult than its offline counterpart. In fact, for $m \geq 2$, there does not exist any optimal online algorithm [3]. To overcome this hardness, Phillips, Stein, Torng, and Wein [8] proposed the use of *resource augmentation* [5]: Given an online algorithm A we determine the speed $s \geq 1$ such that A is optimal on m speed-s processors for any instance that is feasible for m processors of unit speed. We are interested in the smallest s for which

* Research supported by the Indo-German Max Planck Center for Computer Science (IMPECS).

there is an optimal online algorithm. It is known, that any optimal online algorithm needs speed at least $6/5$ [8].

An algorithm is *deadline ordered* if the schedule it yields depends only on the relative ordering of the deadlines of the jobs and not on the actual deadline values. A well-known example of a deadline ordered algorithm is *Earliest Deadline First* (EDF) which at any time schedules the m jobs with the earliest deadline. EDF is optimal on a single machine [2]. On m machines, speed $s = 2-1/m$ is necessary and sufficient to guarantee its optimality [8]. Since its introduction more than a decade ago, this upper bound on the speed requirement for online algorithms has been improved only marginally. Lam and To [6] proposed a more complex deadline ordered algorithm with a speed requirement of $2 - 2/(m + 1)$. They also showed that any deadline ordered online algorithm for m machines needs a speed of at least

$$\alpha_m := \frac{1}{1-\left(1-\frac{1}{m}\right)^m} .$$

For $m = 2$ this quantity equals $4/3$, matching the currently best known upper bound [6], and for arbitrary m it is at most $e/(e-1) \approx 1.58$.

Our main result (Section 3) is a new deadline ordered online algorithm which is optimal with speed α_m. Both, the algorithm and its analysis, build on a simple and elegant online estimate of an optimal schedule (Section 2), proposed in [6]. The matching lower bound in [6] proves that α_m is the exact speed requirement for our algorithm.

We also consider two well-known non-deadline ordered algorithms and provide lower bounds on the speed necessary for them to schedule a feasible instance. Let $p_j(t)$ denote the remaining processing time of job j at time $t \geq r_j$. The laxity of j at time t is defined as $\ell_j(t) = d_j - t - p_j(t)$. The algorithm *Least Laxity First* (LLF) schedules at any point in time m jobs with minimum laxity among the available jobs. LLF is also optimal on a single machine [3], and more generally, it is optimal on m machines when running at speed $2 - 1/m$ [8]. In this paper we provide a lower bound on the speed required (Section 4) by demonstrating a feasible instance for which LLF requires a speed

$$s \geq \frac{1+\sqrt{1+4x^2}}{2x} \quad \text{with } x = \frac{m}{m-1} .$$

This quantity is $(1+\sqrt{17})/4 \approx 1.281$, for $m = 2$, and approaches the golden ratio $(1+\sqrt{5})/2 \approx 1.618$, when m goes to infinity. To the best of our knowledge, this is the first lower bound (beyond the general one) on the speed necessary for LLF. It also shows that our new deadline ordered algorithm outperforms LLF: Indeed, for $m \geq 7$ the lower bound for LLF exceeds the upper bound on the speed required by our new algorithm.

An algorithm that tries to combine features of EDF and LLF is *Earliest Deadline until Zero Laxity* (EDZL) introduced in [1]. At any point in time, EDZL gives highest priority to jobs which cannot be delayed further, i.e, have zero laxity, and other jobs are scheduled in EDF order. This algorithm dominates EDF in the sense that any instance that is schedulable by EDF, is also schedulable by EDZL, whereas the opposite is not true [1,7]. However, it remained open if EDZL is optimal for speed less than $2-1/m$, the speed necessary for EDF. In Section 5, we answer this question negatively by providing a feasible instance on which EDZL fails for speed less than $2 - 1/m$.

2 The yardstick schedule

A key challenge in designing an online algorithm for the deadline scheduling problem is to obtain an estimate of an optimal schedule. Clearly, at any time, we can compute an optimal schedule for the currently known partial instance by solving a maximum flow problem [4]. However, these optimal schedules may differ fundamentally, and it is unclear how an online algorithm can use this information as it cannot change the decisions from the past. Intuitively, we need a less powerful algorithm that computes a simpler optimal solution under some relaxed assumptions. Lam and To [6] proposed a simple and elegant schedule called *yardstick*, which can be constructed online. It has the property that all jobs meet their deadlines but it may not be feasible as it processes a job sometimes simultaneously on multiple machines. The main idea of yardstick is to allow parallelization of a job only if it is *underworked*, i.e., the total amount of processing done on it is smaller than the time period since it was released. We will use yardstick as a reference in the design and analysis of our feasible online algorithm.

The yardstick schedule is constructed as follows. Whenever a new job is released, we consider all unfinished available jobs in increasing order of deadlines and schedule their remaining processing time on the m machines of unit speed. When scheduling job j, we consider the earliest time at which some machine is available and schedule the job on all available machines till it is not underworked any more. Once the total processing done on j equals the time it has been available, we schedule the job on the lowest numbered machine (assuming an arbitrary numbering on the machines) that is available.

Since the yardstick schedule may run a job simultaneously on multiple machines, it does not give a feasible schedule. However, it has the following crucial property.

Lemma 1 ([6]). *If a scheduling instance is feasible, the yardstick schedule completes all jobs before their deadlines.*

In general, it is not necessary to specify for each job the particular machine by which it is processed, but in our case it makes the analysis simpler. In particular, yardstick distributes processing volume in a *staircase profile*: in any time slot, machine $i' > i$ is used only if machine i is also occupied, and between any two release dates the number of machines used is not increasing over time; see Figure 1.

The online algorithm that we present in the following section will at any release time t make reference to the part of the yardstick schedule after time t. This part has a non-increasing staircase profile. Given some (release) time t, let all jobs that have been released by t be indexed in increasing order of deadlines. Recall that this is the order in which yardstick considers jobs. Let Y^j denote the yardstick schedule for jobs $1, 2, \dots, j$ starting at time t. Let $y^j_1, y^j_2, \dots, y^j_i, \dots$ be the time points at which there is a *step* (the total work assigned to the m machines changes) in Y^j. In particular, let $y^j_1 \geq t$ be the first point in time when not all machines are used to their full capacity. Further, let x_j denote the last point in time at which job j is running on multiple machines simultaneously in the yardstick schedule, and let f_j ($f_j > x_j$) denote the time that it finishes processing (see Figure 1). For the sake of readibility we omit in our notation the parameter t since it will always be clear from the context.

The yardstick schedule for jobs released by time t has the following properties.

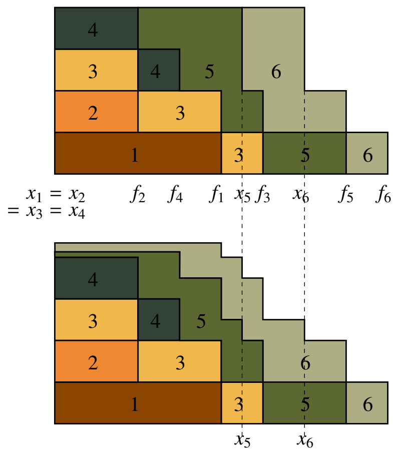

Fig. 1. The yardstick schedule (top) and our schedule (bottom).

(i) If $j < k$, then $x_j < x_k$ although f_j may be greater than f_k.
(ii) Since yardstick runs a job j on multiple machines simultaneously only if it is underworked, we have $f_j \geq r_j + p_j$.
(iii) Since all machines are occupied for all times before x_j, we have $y_1^j \geq x_j$.
(iv) In going from Y^{j-1} to Y^j, all steps before x_j disappear and new steps get created at x_j and f_j (if they do not exist).

3 A best possible deadline ordered online algorithm

We propose a new deadline ordered online algorithm and determine the speed that is sufficient to guarantee that it is optimal.

3.1 Description of the algorithm

Our algorithm mimics the yardstick schedule to a large extent. In particular, it will finish every job by the same time as in the yardstick schedule, which is by Lemma 1 before its deadline. Moreover, our algorithm will have processed at any time at least as much work of any job as the yardstick schedule. Our algorithm will keep up with the yardstick schedule by using faster machines instead of parallelizing jobs. The key idea is that in time periods, in which yardstick schedules a job for some positive amount, our algorithm does not process more than that. In particular when yardstick processes a job which is not underworked and hence uses only one machine, our schedule too processes only one unit of the job in one time unit even though the additional speed would allow for more.

The algorithm works as follows: At any time t when a job is released, we recompute the yardstick schedule. (This is done completely independent of our current online schedule.) Then we consider the set of available unfinished jobs, J_t, in our schedule. We consider jobs in J_t in increasing order of deadlines and assign for each job $j \in J_t$ its remaining processing time $p_j(t)$ at time t in our schedule as follows:

– We schedule one unit of work in each unit-length time slot between x_j and f_j.
– We assign the remaining processing time, $p_j(t) - (f_j - x_j)$, to the slots before x_j with α units assigned to each slot between $s_j := x_j - (p_j(t) - f_j + x_j)/\alpha$ and x_j, where $\alpha \in \mathbb{R},\ 1 \le \alpha < 2$, is a the full speed of the machines.

In this manner we determine for each time slot (after t, as we prove below) the set of jobs to be processed and the extent to which they have to be processed. Notice that we do not allocate particular machines to the jobs. However, the algorithm never assigns more work of an individual job to a time slot than can be processed sequentially. Thus, when assuming integral job parameters and allowing preemption at any time, a round robin like processing yields a schedule with each job using at most one machine at the time and no two jobs using the same machine simultaneously.

Consider the workload profile computed by our algorithm. For the analysis we desire that it has a staircase profile. However the procedure described so far may not satisfy this. In the following we show how to correct this.

At any (release) time t, let the jobs in J_t be indexed in increasing order of deadlines. Note that for both online schedules, yardstick and our schedule, we consider only the part of the schedules after time t, and that both schedules are built by considering jobs (not necessarily the same ones) in the order of increasing deadlines. Similarly as for yardstick we define A^j to be our schedule for jobs $1, 2, \ldots, j \in J_t$ after time t. Let $a_1^j, a_2^j, \ldots, a_i^j, \ldots$ be the time points at which there is a step in A^j. Let $a_1^j \ge t$ be the first point in time when not all machines are used to their full capacity after time t.

Our schedule may not have a staircase profile after the reassignment at some time t, because s_j for some job j may lie between a_i^{j-1} and a_{i+1}^{j-1}. In that case we distribute the part of j scheduled in the interval $[s_j, a_{i+1}^{j-1}]$ uniformly over the interval $[a_i^{j-1}, a_{i+1}^{j-1}]$. This however, may not suffice since the height of the profile in the interval $[a_{i+1}^{j-1}, a_{i+2}^{j-1}]$ might exceed the height of the profile in some of the preceding intervals. If this is the case, we move a suitable amount of job j from $[a_{i+1}^{j-1}, a_{i+2}^{j-1}]$ to the interval preceding it so that these two intervals have the same height in A^j. This process is repeated until we get a staircase profile for A^j. Note that as a consequence of this operation we will be scheduling at most α units of job j in each time slot preceding a_{i+2}^{j-1}.

3.2 Analysis of the algorithm

In this section we show the following main result.

Theorem 1. *The speed $\alpha_m = (1 - (1 - 1/m)^m)^{-1}$ is necessary and sufficient for our algorithm to schedule each job feasibly before its deadline.*

We first show the correctness of our algorithm with respect to time feasibility.

Lemma 2. *If an instance is feasible, then for any job j our algorithm assigns p_j units to feasible time slots between r_j and d_j. In particular, at any time $t \ge r_j$ it (re-)assigns the remaining processing requirement $p_j(t)$ to time slots not earlier than t.*

Proof. First, consider our algorithm without the correction step achieving a staircase profile. At time r_j our algorithm assigns job j to time slots between s_j and f_j. By Lemma 1, $f_j \leq d_j$. The starting time $s_j = x_j - (p_j - f_j + x_j)/\alpha$ is at least $f_j - p_j \geq r_j$ for $\alpha \geq 1$.

Consider some release time $t > r_j$ and the yardstick schedule with the last moments of parallelization x_j (resp. x'_j) and the finishing times f_j (resp. f'_j) of j before (resp. after) rescheduling. Observe that $x'_j \geq x_j$ and $f'_j \geq f_j$. The reason is that yardstick has to schedule more jobs than before and maintains the same scheduling order depending only on deadlines. Our algorithm reassigns the remaining work $p_j(t)$ of j to $[s'_j, f'_j]$. Since this amount has been scheduled in $[t, f_j]$ before, with at most α units per time slot in $[t, f_j]$ and one unit per time slot in $[x_j, f_j]$, our algorithm reassigns it now to later time slots if $f'_j > f_j$ and to a larger amount per time slot if $x'_j > x_j$. Thus, the start time does not decrease, i.e., $s'_j \geq t$. Finally by Lemma 1, we have again $f'_j \leq d_j$ since the instance is feasible.

We have shown that $s_j \geq t$ before the correction step. Now suppose that we must start a job earlier than s_j to obtain a staircase profile. Since $a_i^{j-1} \geq t$, the proof still holds. □

To prove that our algorithm never assigns more work to a time slot than it can process on m fast machines, we proceed as follows. First, we show that at any time the remaining processing time of any job in our schedule is not more than this quantity in the yardstick schedule. Then, we show that at any point in time, our algorithm can distribute the remaining processing time that any job has *in the yardstick schedule* without exceeding the processing capacity under speed α_m.

Lemma 3. *A^j is identical to Y^j for all $t' \geq x_j$.*

Proof. We prove this by induction on j. Suppose the statement is true for all $j \leq k$. Then Y^k and A^k are identical for all t' after x_k and, since $x_{k+1} \geq x_k$, for all $t' \geq x_{k+1}$. The job $k+1$ is scheduled for one unit in each time slot between x_{k+1} and f_{k+1} in both A^{k+1} and Y^{k+1}. This implies both schedules are identical after x_{k+1}. □

Lemma 4. *For any job j and time $t' \leq f_j$, the amount of processing of j remaining at time t' in schedule A^j is less than that remaining at time t' in schedule Y^j.*

Proof. Since schedules A^j and Y^j are identical after x_j (Lemma 3), the remaining processing time of j at any time $t' \geq x_j$ is the same in both schedules.

Since in Y^j, job j is scheduled only after time x_{j-1}, for all times $t' \leq x_{j-1}$ the remaining processing time of j in Y^j is $p_j(t)$ and this is, trivially, at least as large as the remaining processing time of j at any time t' in A^j. Hence we only need to prove the statement for t' in the interval (x_{j-1}, x_j).

In Y^j one or more units of j are scheduled in each time slot between x_{j-1} and x_j. We now consider two cases.

1. At least 2 units of j are scheduled in each slot between x_{j-1} and x_j in Y^j. However, in A^j at most $\alpha < 2$ units of j are scheduled in each slot between x_{j-1} and x_j. Hence for any time t' in the interval (x_{j-1}, x_j), the remaining processing time of j in Y^j exceeds that of j in A^j.

2. Only one unit of j is scheduled in Y^j for some slots in (x_{j-1}, x_j). Since Y^{j-1} has a staircase profile, for all slots between $x_{j-1} = y_1^{j-1}$ and y_2^{j-1}, j must be scheduled for only one unit, while for slots between y_2^{j-1} and x_j, j must be scheduled for at least 2 units.
By the argument in previous case, it follows that for all time $t' \geq y_2^{j-1}$, the remaining processing time of j in Y^j exceeds that of j in A^j. This implies that before time y_2^{j-1}, job j is processed to a larger extent in A^j than in Y^j. From our procedure for scheduling job j it follows that we schedule j for β units in each time slot in the interval (y_1^{j-1}, y_2^{j-1}), where $1 \leq \beta \leq \alpha$. This in turn implies that the amount of j processed before t', $t' \in (y_1^{j-1}, y_2^{j-1})$, is larger in A^j than in Y^j which proves the lemma. □

We discuss some more properties of our schedule with respect to the yardstick schedule. We first argue that in going from A^k to A^{k+1} no new steps are created before x_{k+1}.

Lemma 5. *For every i where $a_i^{k+1} < x_{k+1}$ there is a p such that $a_i^{k+1} = a_p^k$.*

Proof. The proof follows from the way we schedule job $k + 1$. Before time x_{k+1} we schedule job $k + 1$ to an extent of α units in a time slot. Besides, when we redistribute the job over consecutive steps, then no new step is created. □

We use the above lemma for proving the following lemma which will be crucial for our analysis.

Lemma 6. *Consider a job j. In any schedule A^k, $k \geq j$, and for any i such that $a_i^k \leq x_j$, either job j begins at or after a_i^k or it is the case that in all slots between a_i^k and x_j, α units of j is scheduled.*

Proof. Suppose in the schedule A^j, job j begins at time a_p^j. By our method of scheduling j it follows that in all slots between a_{p+1}^j and x_j, α units of j is scheduled. Thus for all $i \leq p$ it is the case that job j begins at or after a_i^j while for $i > p$, all slots between a_i^j and x_j have α units of j.

When we go from schedule A^j to A^{j+1} then, by Lemma 5, we do not create any new steps before x_{j+1}. Since $x_j \leq x_{j+1}$ no new steps are created before x_j either. Further, we do not modify the schedule of job j and so the lemma continues to hold for the schedule A^{j+1} and in a similar manner for all subsequent schedules. □

Now, we are ready to show the correctness of our algorithm with respect to the processing capacity when given speed $\alpha = \alpha_m$. To do so, consider any release time t and the set of available unfinished jobs and their remaining processing times in *the yardstick schedule*. We show that our algorithm applied to these jobs assigns always at most αm units to time slots after t. By Lemma 4 this is only more than our algorithm actually has to schedule.

For the sake of contradiction, assume that this is not the case and let k be the first job for which we fail. This implies that the height of the first step in A^k exceeds αm. In this proof we will assume that $t = 0$. It is straightforward to extend the argument for arbitrary t. Let $p_j(t) = p_j$ be the remaining processing time of j in the yardstick schedule.

Let $a_1^k = z$. Consider the set of jobs scheduled in A^k. We will partition this set into four disjoint subsets. Define B to be the set of jobs, j, for which $f_j < z$ and C as the set of jobs, j, for which $x_j \leq z < f_j$. The remaining jobs are the ones for which $z \leq x_j$ and these we partition into two sets; D is the set of those jobs which begin at or after z in our schedule while E is the set of jobs which begin before z. Counting the total processing time of all jobs in two different ways we get

$$\sum_{j \in B \cup C \cup D \cup E} p_j > m\alpha z + \sum_{j \in C}(f_j - z) + \sum_{j \in D} p_j + \sum_{j \in E}((f_j - x_j) + \alpha(x_j - z)).$$

Note that since a job $j \in E$ begins before $z = a_1^k$ in the schedule A^k, by virtue of Lemma 6, α units of j would have been scheduled in each time slot between z and x_j. Rearranging terms now yields

$$\sum_{j \in B} p_j + \sum_{j \in C}(p_j - f_j + z) + \sum_{j \in E}(p_j - f_j + x_j) > \alpha(mz + \sum_{j \in E}(x_j - z))$$

If α was chosen such that the above inequality is not satisfied, then this would imply that our algorithm never uses more machine capacity than is available. Thus, chosing an α with

$$\alpha > \frac{\sum_{j \in B} p_j + \sum_{j \in C}(p_j - f_j + z) + \sum_{j \in E}(p_j - f_j + x_j)}{mz + \sum_{j \in E}(x_j - z)}$$

guarantees that our algorithm finds a feasible schedule for all jobs when given m machines of speed α. In the following we determine the smallest α that satisfies this condition.

Observe that $\sum_{j \in B} p_j + \sum_{j \in C}(p_j - f_j + z) < mz$.

For $j \in E$ define $b_j = (p_j - f_j + x_j)/z$ and $a_j = x_j/z$. Then

$$0 \leq b_j \tag{1}$$

$$b_j \leq a_j \tag{2}$$

$$a_j \geq 1 \tag{3}$$

Let us number the jobs in E from 1 to $k = |E| \leq m$ in the order of their deadlines. Then for $1 \leq j \leq k$,

$$a_j \geq a_{j-1} + b_j/m \tag{4}$$

where $a_0 = 1$.

Hence it suffices to choose α as the optimal value of the optimization problem

$$\max_{a_i, b_i, 1 \leq i \leq k} \left\{ \frac{m + \sum_{i=1}^k b_i}{m - k + \sum_{i=1}^k a_i} \,\middle|\, (1) - (4) \right\}. \tag{P}$$

Consider an optimal assignment of $b_i, a_i, 1 \leq i \leq k$ for (P). It has the following property.

Lemma 7. *For every $1 \leq i \leq k$, $a_i = a_{i-1} + b_i/m$ and either $b_i = a_i$ or $b_i = 0$.*

Proof. Let p be the largest index for which the statement of the lemma is not true. For every $i \geq p$ we will determine an ϵ_i, δ_i so that the solution remains feasible when for all $i \geq p$ we set

$$a_i \leftarrow a_i + \epsilon_i$$
$$b_i \leftarrow b_i + \delta_i$$

and also when for all $i \geq p$ we set

$$a_i \leftarrow a_i - \epsilon_i$$
$$b_i \leftarrow b_i - \delta_i$$

If the original solution had value X/Y, then the first solution has value $(X + \epsilon)/(Y + \delta)$, where $\epsilon = \sum_{i=p}^{k} \epsilon_i$ and $\delta = \sum_{i=p}^{k} \delta_i$, while the second solution has value $(X-\epsilon)/(Y-\delta)$. If one of these solutions has value greater than that of the original solution then we would have obtained a contradiction.

Otherwise, both solutions have the same value. Our choice of ϵ_i, δ_i will be such that the condition of the lemma remains true for all $i > p$ in both solutions built. Further, in one of the two solutions we will satisfy both conditions for index p (if one was satisfied to begin with) or satisfy one of the conditions for index p (if none were satisfied to begin with). Thus by picking one of these two solutions, which, has the same value as our original solution we get closer to proving the lemma for all indices.

To determine ϵ_i, δ_i for $i = p$ we consider three cases.

$a_p > a_{p-1} + b_p/m, b_p < a_p$: Then $\delta_p = 0$ and $\epsilon_p = \min(a_p - b_p, a_p - (a_{p-1} - b_p/m))$.
$a_p > a_{p-1} + b_p/m, b_p = a_p$: Then $a_p > ma_{p-1}/(m-1)$ and hence $\delta_p = \epsilon_p = a_p - ma_{p-1}/(m-1)$.
$a_p = a_{p-1} + b_p/m, 0 < b_p < a_p$: Then $\delta_p = \min(b_p, a_p - b_p)$ and $\epsilon_p = \delta_p/m$.

The values for $i \geq p + 1$ are determined by considering the following cases

$b_i = a_i$: Then $a_i = a_{i-1} + b_i/m = a_{i-1} + a_i/m$ which implies $a_i = ma_{i-1}/(m-1)$. Hence, $\delta_i = \epsilon_i = m\epsilon_{i-1}/(m-1)$.
$b_i = 0$: Then $\delta_i = 0$ and $\epsilon_i = \epsilon_{i-1}$. □

Let $i_1 < i_2 < \cdots < i_r$ be the indices, i for which $b_i = a_i$. It can be easily shown by induction that $a_0 = a_i = 1, i < i_1$, and $a_{i_1} = m/(m-1) = a_{i_2-1}, a_{i_2} = (m/(m-1))^2 = a_{i_3-1}$, and $a_{i_r} = (m/(m-1))^r$. Thus,

$$\sum_{i=1}^{k} b_i = \sum_{i=1}^{r} \left(\frac{m}{m-1}\right)^i .$$

In an optimal solution to (P), the sum $\sum_{i=1}^{k} a_i$ is minimized. This is the case, when the indices for which $b_i = 0$ are the lowest ones, i.e. $b_i = 0$, $1 \leq k - r$. Then $a_0 = a_1 = \cdots = a_{k-r} = 1$, and thus,

$$\sum_{i=1}^{k} a_i = (k - r) + \sum_{i=1}^{r} \left(\frac{m}{m-1}\right)^i .$$

Hence the value of this solution is

$$\frac{m + \sum_{i=1}^{r} \left(\frac{m}{m-1}\right)^i}{m - k + k - r + \sum_{i=1}^{r} \left(\frac{m}{m-1}\right)^i} = \frac{m\left(\frac{m}{m-1}\right)^r}{m\left(\frac{m}{m-1}\right)^r - r} = \frac{m}{m - r\left(\frac{m-1}{m}\right)^r}.$$

Since $r \leq k \leq m$ and $r(1 - 1/m)^r < m(1 - 1/m)^m$, the above ratio is at most

$$\alpha_m = \frac{1}{1 - (1 - 1/m)^m},$$

and this is our choice of α. For $m = 2$ this quantity equals $4/3$ and for large m, α is at most $e/(e-1)$, since $(1 - 1/m)^m < e^{-1}$. This upper bound combined with the lower bound on the speed requirement of any deadline ordered online algorithm [6] concludes the proof of our main result Theorem 1.

4 A lower bound for LLF

We give a lower bound on the speed that is necessary for LLF to schedule feasible instances. We first give a necessary condition.

Recall that $\ell_j(t)$ is the laxity of job j at time t, and $p_j(t)$ denotes the remaining processing time of j at this time.

Lemma 8. *Let $1 \leq s \leq 2 - 1/m$ be the speed required for LLF to be optimal. Consider an instance that is feasible for m unit speed machines and a time t by which all jobs released before t could have completed in a feasible schedule. If job j has not completed by time t in LLF on m speed-s machines, then*

$$\ell_j(t) \geq \frac{p_j(t)}{s(s-1)}, \tag{5}$$

Proof. Suppose that LLF does not satisfy condition (5) for some job j at time t. We show how to augment the current set of jobs with *blocking jobs* such that LLF will miss the deadline d_j or cannot schedule the blocking jobs feasibly. A set of *blocking jobs* consists of m jobs each having the same size and release time and 0 laxity.

Define the *relative laxity* $\ell_j^r(t)$ of a job as the ratio $\ell_j(t)/p_j(t)$. We first show that if condition (5) is not satisfied, we can decrease the relative laxity arbitrarily. When the relative laxity of the job is sufficiently small, we release m blocking jobs so that no matter how we schedule the jobs, we cannot finish all the jobs by their deadlines.

The procedure for decreasing the relative laxity of some job j is as follows. It is no loss of generality to assume that $p_j(t) = 1$. Since condition (5) is not satisfied, $\ell_j^r(t) = \ell_j(t) = \frac{k}{s(s-1)}$ with $k < 1$. Now we release m blocking jobs each of size q. These jobs will take up q/s time on each machine in the speed-s LLF schedule leaving $q - q/s$ time to process job j. Thus, at time $t' = t + q$, we will have $\ell_j(t') = \ell_j(t) - q/s$ and $p_j(t') = p_j(t) - s(q - q/s)$ whereas an optimal algorithm could have finished all the blocking jobs. We choose q such that the new relative laxity $\ell_j(t')/p_j(t')$ is half the original relative laxity $\ell_j(t)/1$, that is, $q = \frac{2-k}{k(s-1)}$.

Note that we do not release big blocking jobs of size q at one time. We release many small blocking jobs, m at a time, so that each job has laxity less than $\ell_j(t')$. The total size of these small jobs scheduled on a machine is q. This way we can ensure that j is never scheduled in parallel with the blocking jobs. Once these m jobs are finished in the optimal schedule, we release another batch of m blocking jobs.

With the above procedure, we can halve the relative laxity of job j. Repeating this process, we can reduce the relative laxity arbitrarily. Once the relative laxity $\ell_j^r(t'')$ is less than $p_j(t'')/2m$, we release m blocking jobs of size $p_j(t'')$. It is easy to see that these jobs cannot be scheduled within their deadlines. $\square$

Theorem 2. *Let $x = \frac{m}{m-1}$. LLF is not optimal for speed less than*

$$\frac{1 + \sqrt{1 + 4x^2}}{2x},$$

which is $(1 + \sqrt{17})/4 \approx 1.281$ for $m = 2$, and tends to $(1 + \sqrt{5})/2 \approx 1.618$ for $m \to \infty$.

Proof. Let $1 \leq s < 2 - 1/m$ be the speed of the machines available to LLF. We construct a worst case instance consisting of *main* and *blocking* jobs. We have $m + 1$ main jobs with $r_j = 0, p_j = 1, d_j = \frac{m}{m-1}$, for $j = 1, \ldots, m$, and $r_{m+1} = 0, p_{m+1} = \frac{m}{m-1}, d_{m+1} = \left(1 + \frac{1}{s}\right)\left(\frac{m}{m-1}\right)$.

There is a schedule S on m speed-1 machines in which the main jobs are completed by time $t = \frac{m}{m-1}$. Indeed, start the long job k at time 0 and schedule all other jobs in a round robin fashion on the remaining $m - 1$ machines.

In contrast, LLF with speed s schedules the small jobs in a round robin fashion on all m machines and completes them by time $t' = 1/s$. Only then the long job begins processing. To see that, observe that the laxity of the long job $\ell_{m+1}(t'') \geq \ell_1(t'') = \frac{m}{m-1} - \frac{1}{s}$ for any $t'' \leq t'$. The remaining processing time for job $m+1$ at time $t = m/(m-1)$ is $p_{m+1}(t) = \frac{m}{m-1} - \left(\frac{m}{m-1} - 1/s\right)s = 1 - \left(\frac{m}{m-1}\right)(s - 1)$. The lower bound on the speed requirement for LLF now follows directly from Lemma 8. $\square$

5 A lower bound for EDZL

We show that the speed requirement of EDZL to guarantee an optimal schedule for a feasible instance is no less than the one for EDF.

Theorem 3. *EDZL is not optimal for speed $s = 2 - \frac{1}{m} - \epsilon$ for any $\epsilon > 0$.*

Proof. We first give the proof for $m = 2$ and show later how this can be generalized to arbitrary $m > 2$. At time $t = 0$, three jobs are released: two of them have size L and deadline $2L$. The remaining job (call this job j) has size $2L$ and deadline $3L/s$. At $t = 2L$, two jobs of size 1 and deadline $2L + 1$ are released. Assume that $L \gg 1$ so that $3L/s > xL + 1$.

We first note that there is a feasible schedule for this instance. Schedule job j on one machine and the other jobs in a round robin fashion on the other machine. All the jobs are finished by time $2L$. Then schedule the newly arrived jobs on separate machines.

EDZL will schedule the size L jobs on 2 machines till time L/s. At $t = L/s$, the size L jobs will be finished and job j has zero laxity. Since we assume that $L \gg 1$, one machine must be assigned to this job for the remainder of the schedule. Clearly the newly released jobs at time $t = 2L$ cannot both be scheduled on the other machine. Thus EDZL is not optimal for $s = 3/2 - \epsilon$ for $m = 2$.

For general m, let $x = \frac{m}{m-1}$. We release $m + 1$ jobs at time 0 with one job of size xL_1 and deadline $(x + 1)L_1/s$ and the remaining m jobs of size L_1 and deadline xL_1. Similar to the above analysis, at time $t = xL_1$, all jobs will be finished by an optimal offline schedule while there is one tight job in EDZL. We choose L_1 large enough so that one machine is tied up to this job for the rest of the schedule. We repeat the construction with $m-1$ machines with some large enough L_2 and continue until we reach the case $m = 2$ in which a job will fail its deadline. $\square$

6 Concluding remarks

As our main result we have introduced a new online algorithm which is optimal for speed $\alpha_m \leq e/(e-1) \approx 1.58$. This is the first significant improvement since the seminal results in [8]. Our algorithm is best possible in the class of deadline ordered algorithms with respect to speed resource augmentation. Nevertheless, this does not generally rule out online algorithms that are optimal for less speed. However, we showed that our algorithm outperforms all algorithms, deadline ordered and non-deadline ordered, for which provable upper bounds are known in the literature.

References

1. S. Cho, S. K. Lee, S. Ahn, and K.-J. Lin. Efficient real-time scheduling algorithms for multiprocessor systems. *IEICE Transactions on Communications*, E85-B(12):2859–2867, 2002.
2. M. L. Dertouzos. Control robotics: The procedural control of physical processes. In *IFIP Congress*, pages 807–813, 1974.
3. M. L. Dertouzos and A. K. Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, 1989.
4. W. A. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21(1):177–185, 1974.
5. B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM*, 47(4):617–643, 2000.
6. T. W. Lam and K.-K. To. Trade-offs between speed and processor in hard-deadline scheduling. In *Proc. of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 623–632, 1999.
7. M. Park, S. Han, H. Kim, S. Cho, and Y. Cho. Comparison of deadline-based scheduling algorithms for periodic real-time tasks on multiprocessor. *IEICE Transactions*, 88-D(3):658–661, 2005.
8. C. A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. *Algorithmica*, 32(2):163–200, 2002.