Trade-offs between Speed and Processor in Hard-deadline Scheduling

Tak Wah Lam* Kar Keung To†

Abstract

This paper revisits the problem of on-line scheduling of sequential jobs with hard deadlines in a preemptive, multiprocessor setting. An on-line scheduling algorithm is said to be *optimal* if it can schedule any set of jobs to meet their deadlines whenever it is feasible in the off-line sense. It is known that the earliest-deadline-first strategy (EDF) is optimal in a one-processor setting, and there is no optimal on-line algorithm in an m-processor setting where $m \geq 2$. Recent work [Phillips *et al.* STOC 97] however reveals that if the on-line algorithm is given faster processors, EDF is actually optimal for all m (e.g., when $m = 2$, it suffices to use processors 1.5 times as fast).

This paper initiates the study of the trade-off between increasing the speed and using more processors in deriving optimal on-line scheduling algorithms. Several upper bound and lower bound results are presented. For example, the speed requirement of EDF can be reduced to $2 - \frac{1+p}{m+p}$ when it is given $p \geq 0$ extra processors. The main result is a new on-line algorithm which demands less speedy processors so as to attain optimality (e.g., when $m = 2$, the speed requirement is $1\frac{1}{3}$) and admits a better speed-processor trade-off than EDF (e.g., when $m = 2$ and $p = 1$, the speed requirement is 1.2). In general, no optimal algorithm exists when the speed factor is less than $1/(2\sqrt{2+p/m} - 2)$.

1 Introduction

Problems of on-line scheduling have a rich literature (see e.g., [1, 2, 7, 11, 13] for some recent results). In this paper we consider a classical scheduling problem encountered in a hard-deadline real-time environment. There is a pool of m identical processors. At any time a job can be released, which is sequential in nature (i.e. to be run on at most one processor at any time), and must be completed before a certain deadline. The release time of a job, as well as the required amount of work and the deadline, is known only when the job is released. This model has been considered by many authors (see e.g., [9, 10, 12, 13]) for scheduling with deadlines. Our goal is to devise an on-line algorithm to schedule such jobs so that their deadlines can all be met. We allow preemptive scheduling, which is commonly used in computer systems. Not every set of jobs can be scheduled to meet their deadlines. We expect an on-line algorithm to meet the deadline requirements whenever it is feasible in the off-line sense. Such an on-line algorithm is said to be *optimal*.

When there is only one processor for scheduling, the earliest-deadline-first strategy (EDF) gives an optimal algorithm [4]. However, when there are two or more processors, it is known that no on-line algorithm is optimal [5]. Recently, there are studies on the effect of giving the on-line algorithm faster processors in different scheduling problems [3, 6, 10, 13]. Intuitively, using faster processors compensates the on-line algorithm for the lack of future information. In the following, we assume that the off-line algorithm is given processors of speed-1 and the on-line algorithm is given processors of speed-s for some $s \geq 1$ (precisely, a speed-s processor can process x units of work in x/s units of time). Regarding hard-deadline scheduling in particular, Phillips *et al.* [13] showed that, when using processors of speed-$(2 - 1/m)$, EDF is actually optimal.[1] That means, for a two-processor system, the speed requirement is 1.5. Phillips *et al.* [13] also showed a lower bound of 1.2 for all $m \geq 2$.

Another way to facilitate the on-line algorithm is to use more processors. However, due to the sequential nature of the jobs, more processors may not improve the performance. It is known that EDF cannot achieve optimality even if it is allowed to use up to $O(m)$ speed-1

*Department of Computer Science, The University of Hong Kong. Email: twlam@cs.hku.hk

†Department of Computer Science, The University of Hong Kong. Email: kkto@cs.hku.hk

[1]Phillips also showed that the least-laxity-first (LLF) strategy is optimal when given speed-$(2 - 1/m)$ processors. However, LLF may schedule jobs to migrate among processors infinitely frequently, whereas EDF (as well as our new algorithm) requires only $O(n)$ migrations for a set of n jobs.

processors [13]. As far as we know, no on-line algorithm using $O(m)$ speed-1 processors is optimal.

This paper initiates the study of the trade-off between increasing the speed and using more processors so as to attain optimality. Several upper bound and lower bound results are presented, illustrating to what extent using more processors can reduce the speed requirement.

For EDF, we show that the speed requirement is reduced to $2 - \frac{1+p}{m+p}$ when there are $p \geq 0$ additional processors. In general, when m is large, we find that no optimal algorithm exists when the processor speed is less than $1/(2\sqrt{2+p/m}-2)$ (which is roughly equal to 1.207 when $p = 0$). The main result in this paper is a new scheduling algorithm which demands a smaller speed requirement to attain optimality and admits a better speed-processor trade-off than EDF. Precisely, when there are $p \geq 0$ extra processors, the speed requirement is $2 - \frac{2(m-1)+mp}{(m+1)(m-1)+mp}$ (i.e. $2 - \frac{2}{m+1}$ if $p = 0$). For example, when $m = 2$ and there is no extra processor, the speed requirement is $1\frac{1}{3}$; when $m = 2$ and $p = 1$, the speed requirement is 1.2. The improvement is rooted at a simple yardstick schedule which results from an attempt to estimate the optimal off-line schedule.

Like EDF, our new algorithm depends on the relative ordering of the deadlines of jobs instead of their actual values. We call such algorithms *deadline-ordered*. In this paper we also show improved lower bounds on the speed and processor requirements of such kind of algorithms. When p extra processors are available, the speed requirement is at least $1 \,/\, (1 - (\frac{m-1}{m})^m + \frac{p}{m}(\frac{m-1}{m})^{m-1})$. When $m = 2$ and $p = 0$, this lower bound is equal to $1\frac{1}{3}$; therefore, our algorithm is actually the best possible deadline-ordered algorithm in this case. Our work also implies that using speed-1 processors, any algorithm requires at least $m - 1$ extra processors to achieve optimality. This result should be compared with the previously known lower bound for general algorithms [13], which states that at least $m/4$ extra processors must be used to achieve optimality.

As a summary, we illustrate in Figure 1 the speed-processor trade-off achieved by EDF and our new algorithm, as well as the lower bounds for deadline-ordered algorithms and general algorithms.

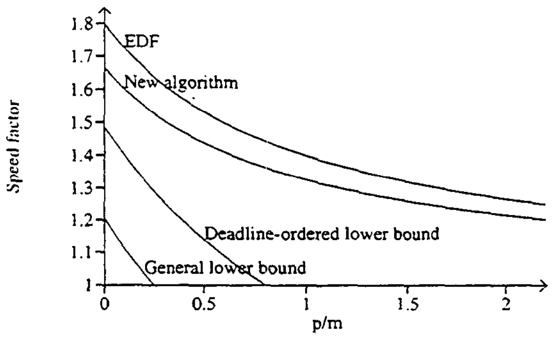

Figure 1: Trade-offs between speed and processor when $m = 5$.

Related work: In the literature, the study of using faster processors to enhance on-line scheduling algorithms covers not only hard-deadline systems, but also some less stringent requirements. Based on a setting similar to the one considered in this paper, Kalyanasundaram and Pruhs [10] have studied one-processor scheduling with an objective to maximize the benefits earned from jobs that meet the deadlines, and Phillips *et al.* [13] have studied the multiprocessor scheduling in view of average response time. There are also a number of interesting results based on a single-processor system in which jobs have unknown processing time [3, 10]; the primary concern is again on average response time. Edmonds [6] recently extended these works to a multiprocessor setting in which jobs are parallelizable (i.e., jobs can be speeded up by using more processors).

Organization: The remainder of this paper is organized as follows. Section 2 serves as a warm-up, showing that using more processors, EDF demands less on processor speed to achieve optimality. Section 3 shows a lower bound on the speed requirement of any optimal on-line algorithm using $m+p$ processors. Section 4 describes a simple way to estimate the optimal off-line schedule, which is then used in Section 5 to derive a new on-line algorithm attaining the improved results claimed earlier. Section 6 shows new lower bounds for the class of deadline-ordered on-line algorithms. Before we leave this section, let us clarify how an on-line scheduling algorithm operates. The algorithm is invoked whenever an *interrupt* occurs. An interrupt is either triggered by the release of a job, or preset by the algorithm itself in some previous invocation. The output of the algorithm is a mapping from the jobs to the processors. The algorithm is said to be optimal if it never misses a deadline for any feasible job set, that is, any job set which admits a schedule which meets all the deadlines using m speed-1 processors.

2 Speed-processor trade-off for EDF

With EDF, job assignment is done only when a job is released or completed. Basically, whenever the number of remaining jobs does not exceed the number of processors, every job is assigned to a distinct processor; otherwise, only those with the earliest deadlines are assigned. In this section, we extend the work of Phillips *et al.* [13] to show that the speed requirement for EDF to achieve optimality can be lowered when more processors are used. In particular, we prove that EDF is optimal if it is given $m + p$ speed-$(2 - \frac{1+p}{m+p})$ processors, where p is any nonnegative integer (Corollary 2.1). We also show that this trade-off result is tight at the end of this section (Lemma 2.2).

The following lemma generalizes a property given in [13] about the work done in accordance with EDF, as well as any "busy" scheduling algorithm (i.e. any algorithm that produces a schedule satisfying that whenever there is an idling job that is not yet completed, all processors must be working on other jobs). Using this property, we obtain the trade-off result for EDF.

LEMMA 2.1. *Consider any algorithm $\mathcal{A}$ using m speed-1 processors, and any busy scheduling algorithm $\mathcal{A}'$ using $m + p$ speed-s processors, where $s \geq 2 - \frac{1+p}{m+p}$. Suppose $\mathcal{A}$ and $\mathcal{A}'$ are each used to schedule a job set $\mathcal{L}$. At any time t, let $\mathcal{A}(\mathcal{L}, t)$ denote the total work done on $\mathcal{L}$ up to t with $\mathcal{A}$, and similarly $\mathcal{A}'(\mathcal{L}, t)$ with $\mathcal{A}'$. Then $\mathcal{A}'(\mathcal{L}, t) \geq \mathcal{A}(\mathcal{L}, t)$.*

Proof. We prove the lemma by contradiction. Assume that at some time t, $\mathcal{A}'(\mathcal{L}, t) < \mathcal{A}(\mathcal{L}, t)$. Then there must be a job J such that $\mathcal{A}'(\{J\}, t) < \mathcal{A}(\{J\}, t)$. Denote r_J as the release time of J. Without loss of generality, we assume that $\mathcal{A}'(\mathcal{L}, r_J) \geq \mathcal{A}(\mathcal{L}, r_J)$ (otherwise we replace t by r_J). Consider the period of time between r_J and t. Let x be the amount of time when $\mathcal{A}'$ uses all processors available, and let y be the rest of the time. Since $\mathcal{A}'$ is a busy algorithm, whenever $\mathcal{A}'$ is not using all processors, J must be worked on by one processor. Therefore,

$$sy \leq \mathcal{A}'(\{J\}, t) < \mathcal{A}(\{J\}, t) \leq x + y.$$

Consider the work done for all jobs during the time interval (r_J, t). We have

$$\begin{aligned} s((m+p)x + y) &\leq \mathcal{A}'(\mathcal{L}, t) - \mathcal{A}'(\mathcal{L}, r_J) \\ &< \mathcal{A}(\mathcal{L}, t) - \mathcal{A}(\mathcal{L}, r_J) \leq m(x + y). \end{aligned}$$

Combining the above inequalities, we have $s(m+p)(x+y) < (2m + p - 1)(x + y)$, which contradicts the bound of s. □

COROLLARY 2.1. *Let p be any nonnegative integer. EDF, when given $m + p$ speed-s processors where $s \geq 2 - \frac{1+p}{m+p}$, is optimal.*

Proof. Suppose, for the sake of contradiction, that there exists a feasible job set $\mathcal{L}$ for which EDF causes the deadline of a job J to be missed. Denote $\Delta(J)$ as the set of jobs in $\mathcal{L}$ with deadlines earlier than J. Note that $\Delta(J)$ can be scheduled by some off-line algorithm $\mathcal{A}$, and using EDF to schedule $\Delta(J)$ would again cause the deadline of J to be missed. At the deadline of J, all jobs are completed with $\mathcal{A}$ but not with EDF. This contradicts Lemma 2.1. □

LEMMA 2.2. (Tightness) *Let p be any nonnegative integer. EDF, when given $m + p$ speed-s processors where $s < 2 - \frac{1+p}{m+p}$, is not optimal.*

Proof. It suffices to show a feasible job set for which EDF fails to schedule all the jobs to meet the deadlines. Consider a set of $m + p + 1$ jobs, all released at time 0. Among them, $m + p$ jobs are "short", with required work $m - 1$ and deadline $m + p$. The last job is "long", with required work $m + p$ and with deadline $m + p + \epsilon$ for some $\epsilon > 0$. This job set is feasible as all deadlines can be met by a simple schedule: Allocate a processor solely for the long job, and allocate other processors for the small jobs in a round-robin fashion. With EDF, the $m + p$ processors are each allocated to the short jobs until all of them are completed. The time for the long job to complete is $(m-1+m+p)/s$. Since $s < 2 - \frac{1+p}{m+p}$, $(m-1+m+p)/s > m+p+\epsilon$ for some ϵ chosen suitably. So EDF misses the deadline of the long job. □

3 A lower bound for trading processor for speed

In this section, we derive a lower bound for the speed requirement of any on-line algorithm using $m + p$ processors. We show that such algorithms need speed-$\frac{km+m^2}{k^2+m^2+pm}$ processors to achieve optimality, where k is any integer between 0 and m. In particular, choosing $k = m/2$, we obtain a speed lower bound of $\frac{6}{5+4p/m}$. This result implies the lower bound of 1.2 given in [13] for the case using no extra processor. When m is large, we can choose a suitable k so that the speed requirement approaches $1/(2\sqrt{2 + p/m} - 2)$. In the special case when $p = 0$, the bound is approximately 1.207, slightly improving the previous lower bound.

LEMMA 3.1. *No on-line algorithm using $m + p$ speed-s processors is optimal when $s < \frac{km+m^2}{k^2+m^2+pm}$, where k is any integer between 0 and m.*

Proof. To show this lower bound, we consider the following repetitive list of jobs. Assume that the time frame is divided into iterations, each of length 1. At the beginning of each iteration, the following jobs are released.

- m jobs—required work: $1 - k/m$; deadline: at the end of the iteration;
- k jobs—required work: 1; deadline: at the end of the next iteration.

The number of iterations is determined by the adversary. In the final iteration, there are m additional jobs. They are released exactly after $1 - k/m$ units of time has elapsed; each job requires k/m units of work and has deadline at the end of this iteration. It is easy to see that an off-line algorithm (using m speed-1 processors) can schedule all the jobs to meet their deadlines.

We define the *critical moment* of an iteration to be the time when $1 - k/m$ units of time has elapsed. An on-line algorithm (using $m+p$ speed-s processors) does not know whether an iteration is the final one or not until it sees the additional jobs released at the critical moment. Just before the critical moment, let w be the amount of remaining work due to jobs with deadlines at the end of the current iteration. Note that w cannot be too large. Otherwise, if the current iteration is indeed the final one, the additional jobs cannot be completed in time. More precisely, the algorithm must maintain $k + w \le s(m + p)\frac{k}{m}$. In each iteration other than the final one, the amount of work that can be done starting from the critical moment is at most $w + sk(k/m)$. Thus the maximum amount of work that can be done in each iteration is

$$\begin{aligned}&(1 - \frac{k}{m})s(m + p) + w + sk(\frac{k}{m})\\ \le\;& (1 - \frac{k}{m})s(m + p) + s(m + p)\frac{k}{m} - k + sk\frac{k}{m}\\ =\;& s(p + m + k^2/m) - k.\end{aligned}$$

Suppose the speed of the processors are $(1-\epsilon)(km+m^2)/(k^2+m^2+pm)$ for some $\epsilon > 0$. Then the maximum amount of work that can be done in each iteration is

$$\begin{aligned}&s(p + m + k^2/m) - k\\ =\;& (1 - \epsilon)\left(\frac{km + m^2}{k^2 + m^2 + pm}\right)(p + m + k^2/m) - k\\ =\;& m - (k + m)\epsilon.\end{aligned}$$

Note that the total amount of work due to the jobs released in each non-final iteration is exactly m. In other words, the on-line algorithm fails to schedule at least $(k+m)\epsilon$ units of work after the first iteration, and $2(k + m)\epsilon$ after the second iteration. At the end of the $(\lfloor k/((k + m)\epsilon)\rfloor + 1)$-st iteration, the on-line algorithm has accumulated more than k units of work not yet scheduled. Since the amount of work with deadline later than the end of that iteration is only k, at least one of the jobs with deadline at the end of that iteration misses its deadline. □

COROLLARY 3.1. *When m is even, no on-line algorithm using $m + p$ speed-s processors is optimal when $s < 6/(5 + 4p/m)$.*

Proof. Putting $k = m/2$ in Lemma 3.1 yields the corollary directly. □

When m is large enough, we can improve this bound by choosing k more carefully. That is, when m is arbitrarily large, we can choose k arbitrarily close to $\sqrt{2m^2 + pm} - m$. This implies that the speed requirement for an optimal algorithm using $m + p$ processors is at least $1/(2\sqrt{2 + p/m} - 2)$.

4 Yardstick schedule versus off-line schedule

EDF is primarily based on a greedy strategy to drive its speedy processors to meet the deadlines. It does not care how the jobs are actually scheduled using m speed-1 processors. Our new algorithm, called FR, performs better by making reference to a *yardstick* schedule using m speed-1 processors. This yardstick schedule meets the deadlines of any feasible job set, although it is not a realistic schedule and cannot be used directly.

Note that an on-line algorithm does not know the jobs in advance. At a particular time, a natural yardstick it can refer to is the optimal off-line schedule (using m speed-1 processors) for the jobs released so far. Yet polynomial time algorithms for computing the optimal off-line schedule is known only in the special

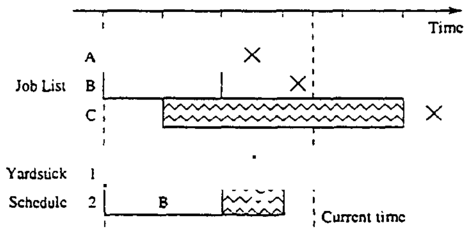

Figure 2: An example of yardstick schedule using 2 processors. Crosses in the figure denote deadlines of the jobs. Jobs A and B are released at time 0, while Job C at time 1. Note that C runs on both processors for 1 unit of time in the yardstick schedule.

case when $m = 2$ [8]. Moreover, whenever a new job is released, the optimal schedule may change drastically, which then has little value to the on-line algorithm since it cannot roll back its decision.

To arrive at a good yardstick schedule, we make an unrealistic assumption that when a job J is under-worked (i.e., the work done on J is smaller than the period of time since J is released), J can be speeded up by running on more than one processor simultaneously. With this assumption, we can extend EDF to obtain an on-line algorithm, denoted YS below, that can produce a good yardstick schedule using m speed-1 processors. YS is invoked when a job is released or completed. YS always considers the remaining jobs in ascending order of the deadlines. Before scheduling a job J, YS takes an extra step to examine whether J is under-worked. If not, YS simply allocates one processor for J; if so, YS allocates all remaining processors to work on J. In the latter case, YS will be invoked again as soon as J becomes no longer under-worked. See Figure 2 for an example.

Note that YS is not a "real" scheduler as it depends on an unrealistic assumption. Yet YS is optimal since processors are kept to be busy as far as possible in the following sense.

LEMMA 4.1. *At any time, the total amount of work scheduled for all released jobs by any real scheduler using m speed-1 processors cannot exceed that by YS.*

Proof. Suppose the contrary holds for some time period with respect to a real scheduler. Then within the time period, there must be a time when the real scheduler schedules more processors to work than YS. At this time YS must have some processors idle. By definition, this can happen only if every released job that is not completed is scheduled on a processor and the amount of work done for these jobs must equal to the amount of time they had released. In other words, the amount of work done for any job by the real scheduler cannot exceed that by YS, contradicting the proposition that the real scheduler has done more work over all released jobs. □

LEMMA 4.2. *Using m speed-1 processors, YS produces a schedule which does not miss any deadline for any feasible job set.*

Proof. Suppose that there is a feasible job set $\mathcal{L}$ in which some jobs miss their deadlines under the schedule produced by YS. Among these jobs, let J be the job with the earliest deadline. Denote $\Delta(J)$ as the subset of $\mathcal{L}$ in which jobs have deadlines earlier than the deadline of J. Consider the scheduling of $\Delta(J)$ by YS. Since YS determines the schedule of a job before examining any job with later deadline, the scheduling of jobs in $\Delta(J)$ is the same as the scheduling of $\mathcal{L}$. Therefore, J still misses its deadline when YS is used to schedule $\Delta(J)$. Also, $\Delta(J)$ is feasible since it is a subset of a feasible job set. This however contradicts Lemma 4.1 at the deadline of J, since all jobs are completed in the optimal schedule but not by YS. □

The assumption made by YS actually does not give it too much power. In fact, we can prove that EDF using m speed-$(2 - \frac{1}{m})$ processors can match the progress of YS on every job at any time. In §5, we see that YS serves as a yardstick for our new algorithm FR, which uses less speedy processors to match the progress of YS and completes every job no later than YS does.

5 FR—a less speed-demanding algorithm

By simulating the on-line algorithm YS, we can modify EDF to adjust dynamically so as to achieve optimality with less speedy processors. The new algorithm is called FR. The reduced speed requirement is $2-\frac{2}{m+1}$. FR also admits a trade-off between speed and processor. We show that if $p \geq 0$ extra processors are available, the speed requirement of FR becomes $2 - \frac{2(m-1)+mp}{(m+1)(m-1)+mp}$ (§5.2). For example, when $m = 2$ and $p = 0$, the speed requirement is $1\frac{1}{3}$; when $m = 2$ and $p = 1$, the speed requirement is 1.2.

In the full paper we will show that both results are tight, in the sense that FR is not optimal with slower processors. The number of migrations induced by FR is linear in terms of the number of jobs, comparable to that of EDF (§5.3).

As mentioned in the previous section, EDF using m speed-$(2 - \frac{1}{m})$ processors can match the progress of YS which uses m speed-1 processors. In fact, in most cases, EDF schedules a job to work much faster than YS, and completes the job earlier. Notice that YS meets the deadlines of all jobs and there is no need to work faster than YS. The key idea of FR is to use a less greedy strategy for jobs whenever these jobs have been "over-scheduled", as compared with the yardstick schedule.

Roughly speaking, FR attempts to apply EDF to schedule the jobs. This allows FR, using speed-s processors for some $s \geq 2 - \frac{2}{m+1}$, to outperform any speed-1 algorithm (and YS) on jobs with earlier deadlines as quickly as possible. However, when the remaining work of such a job J, denoted by $W_{FR}(J)$, becomes only a small fraction (precisely, s/m) of the remaining work of J in the yardstick schedule, FR deliberately slows down its execution as follows. Whenever YS allocates $k \geq 1$ processors to J, FR allocates k/m of a processor for it. Thus, the ratio between $W_{FR}(J)$ and $W_{YS}(J)$ remains to be s/m until J completes. As a result of this slowdown, jobs with later deadlines can be started earlier than in EDF, and they can eventually be completed by FR using less speedy processors.

The algorithm FR is illustrated in Algorithm 1. Once a job is released, it is said to be in "`full`" mode. At the time when $W_{FR}(J) = (s/m)W_{YS}(J)$, the job is said to be in "`reduced`" mode. Jobs in `reduced` mode are always allocated in a specific processor, which is denoted by P_l. In §5.1 we see that speed-$(2 - \frac{2}{m+1})$ processors are sufficient to guarantee that each job eventually switches to `reduced` mode, and completes exactly when it completes in the yardstick schedule.

It is worth-mentioning that FR, like some other well-known scheduling algorithms such as Balance and Equi-partition (see e.g., [3, 6, 10]), takes advantage of the time-sharing capability of processors. Yet FR only needs time-sharing in at most one processor (i.e., P_l). For convenience, the discussion in Algorithm 1 allows P_l to be time-shared by up to m jobs. In the full paper, we will give a small modification to FR, with which at most two jobs are scheduled to time-share P_l. This modification is not only of theoretical interest. In practice, time-sharing among many jobs causes a lot of overheads.

5.1 Optimality In the remainder of this section we show that FR is optimal when given m speed-s processors, where $s \geq 2 - \frac{2}{m+1}$. With the optimality of YS (Lemma 4.2), it suffices to prove the following lemma:

LEMMA 5.1. *Consider the scheduling of any feasible job set $\mathcal{L}$ using FR, as well as that using YS. At any time t, for any job $J \in \mathcal{L}$, $W_{FR}(J) \leq W_{YS}(J)$.*

Proof. We prove this lemma by contradiction. Without loss of generality, suppose the job J_0 with the latest deadline in $\mathcal{L}$ is the only job failing to satisfy the lemma (otherwise we find the job J with the earliest deadline violating the lemma, and replace $\mathcal{L}$ with the subset of $\mathcal{L}$ with deadlines no later than J). Let r denote the release time of J_0. Let t be the first time such that $W_{FR}(J_0) = W_{YS}(J_0)$ at t, and $W_{FR}(J_0) > W_{YS}(J_0)$ right after t. That is, YS makes more progress on J_0 than FR starting from t. Note that J_0 must be in `full` mode at time t (because once a job J has switched to `reduced` mode, $W_{FR}(J) = (s/m)W_{YS}(J)$). The only way YS makes more progress on J_0 from t onward is to use multiple (≥ 2) processors for J_0 at t. Therefore, J_0 is under-worked by YS at t. We observe a number of interesting properties:

⟨1⟩ The amount of work YS has scheduled on J_0 up to t, which is equal to the amount of work FR has scheduled on J_0, is strictly less than $t - r$.

⟨2⟩ At t, every job other than J_0 is either already completed by YS (as well as by FR), or not under-worked and selected for execution by YS. Otherwise, YS cannot selects J_0—the job with the latest deadline—for execution. We call these two categories of jobs $\mathcal{C}$ and $\mathcal{R}$ respectively. Note that $|\mathcal{R}| \leq m - 2$.

⟨3⟩ Let t' be any time before t. Consider the amount of work scheduled for J_0, as well as any job in $\mathcal{C}$, during the period of time (t', t). The amount of work scheduled with YS is at least that with FR. This is because at t, $W_{YS}(J_0) = W_{FR}(J_0)$ and $W_{YS}(J) = W_{FR}(J) = 0$ for every job J in $\mathcal{C}$; and at t', $W_{YS}(J) \geq W_{FR}(J)$ for every job J.

We partition the time period from r to t according to the processor share FR allocates to J_0. For each $0 \leq i \leq m$, we denote x_i as the total length of time periods during which FR allocates i/m of a processor

Algorithm 1 FR

```
1: Update $\mathtt{mode}(J)$ for each job $J$.
2: Simulate YS.
3: for all jobs $J$ which YS schedules $k \geq 1$ processors to work on $J$ do
4:   if $\mathtt{mode}(J) = \mathtt{full}$ then
5:     Schedule $J$ to one processor, using processors other than $P_l$ if possible.
6:   else
7:     Schedule $J$ to $k/m$ of $P_l$.
8: while some processor other than $P_l$ is not used and some full mode job is not scheduled do
9:   Schedule the job with the earliest deadline among these jobs to that processor.
10: if $P_l$ is not fully used and some full mode job is not scheduled then
11:   Schedule the job with the earliest deadline among these jobs to all remaining share of $P_l$.
12: Preset an interrupt to occur at the earliest of the following: an interrupt preset by YS, the first time when a job would
    complete, and the first time when a job would change mode.
```

to J_0. The amount of work FR scheduled on J_0 up to time t is $\sum_{i=0}^{m} s(i/m)x_i$. By ⟨1⟩, this is strictly less than $t-r = \sum_{i=0}^{m} x_i$. We thus obtain an upper bound on the amount of time when FR uses one full processor for J_0.

$$(5.1) \qquad (s-1)x_m < \sum_{i=0}^{m-1}\left(1-\frac{si}{m}\right)x_i$$

On the other hand, we observe that FR has a tendency to allocate a full processor to J_0 and thus x_m has a sufficiently large lower bound, contradicting the upper bound above.

For $0 \leq j \leq |\mathcal{R}|$, define r_j as the first time after r when j or more jobs have been releases. For $|\mathcal{R}|+1 \leq j \leq m-1$, we define r_j to be t. Note that $r_0 = r$. Denote $x_{i,j}$ as the total length of time periods in (r_j, r_{j+1}) during which FR allocates i/m of a processor to J_0. Note that $x_i = \sum_{j=0}^{m} x_{i,j}$. In the following discussion, we inductively show for $\phi = m-1$ down to 0 a relation concerning the period of time (r_ϕ, t):

$$(5.2) \qquad \sum_{i=0}^{m-1}\left(1-\frac{si}{m}\right)\sum_{j=\phi}^{m-1} x_{i,j} \leq (s-1)\sum_{j=\phi}^{m-1} x_{m,j}.$$

When $\phi = 0$, (5.2) leads to:

$$\sum_{i=0}^{m-1}\left(1-\frac{si}{m}\right)x_i \leq (s-1)x_m.$$

We thus have a contradiction with (5.1). This shows that J_0 does not exist and Lemma 5.1 holds. □

The inductive proof of (5.2) goes from the trivial case where $\phi = m-1$, in which all $x_{i,m-1}$ are zero (since $|\mathcal{R}| \leq m-2$). Assuming that (5.2) holds for $\phi_0 \leq m-1$, we consider the case for $\phi_0 - 1$. We add up (5.2) for all $\phi \geq \phi_0$ to produce

$$(5.3) \qquad \sum_{i=0}^{m-1}\left(1-\frac{si}{m}\right)\sum_{j=\phi_0-1}^{m-1}(j-\phi_0+1)x_{i,j} \leq (s-1)\sum_{j=\phi_0-1}^{m-1}(j-\phi_0+1)x_{m,j}.$$

We analyze how much work YS has performed on all jobs, i.e. $\{J_0\}\cup\mathcal{R}\cup\mathcal{C}$, during the time period (r_{ϕ_0-1}, t). With respect to this period of time, let w_0 and w_1 be the amount of work done for jobs in $\mathcal{C}$ with YS when J_0 runs on a partial and full processor respectively according to FR. By ⟨3⟩, the amount of work done for J_0 with YS is at least that with FR, i.e. $\sum_{i=0}^{m}\sum_{j=\phi_0-1}^{m-1} s(i/m)x_{i,j}$. The amount of work done with YS for jobs in $\mathcal{C}$ is $w_0 + w_1$. For a job in $\mathcal{R}$ released at r_j on or after r_{ϕ_0-1}, exactly $t - r_j$ work is done since the job is not under-worked at t. Similarly, for a job in $\mathcal{R}$ released before r_{ϕ_0-1}, at least $t - r_{\phi_0-1}$ work is done. This leads to at least $\sum_{i=0}^{m}\sum_{j=\phi_0-1}^{m-1} jx_{i,j}$ work done for jobs in $\mathcal{R}$. On the other hand, YS can do at most $m(t-r_{\phi_0-1}) = m\sum_{j=\phi_0-1}^{m-1} x_{i,j}$ work during the period of time, producing the following relationship.

$$(5.4) \qquad \sum_{i=0}^{m}\sum_{j=\phi_0-1}^{m-1}\frac{si}{m}x_{i,j} + \sum_{i=0}^{m}\sum_{j=\phi_0-1}^{m-1} jx_{i,j} + w_0 + w_1 \leq m\sum_{i=0}^{m}\sum_{j=\phi_0-1}^{m-1} x_{i,j}$$

We obtain another relationship concerning the time period (r_{ϕ_0-1}, t) when we study the work on $\mathcal{C}$ with FR. Whenever FR uses less than one processor for J_0, no processor can be idle (since J_0 is still in full mode). Between r_j and r_{j+1}, only jobs released on or before r_j can be scheduled since other jobs are released on or after r_j. So only j processors can be used for jobs in $\mathcal{R}$. Exactly i/m processors are used for J_0 for a period of length $x_{i,j}$. All the remaining processors must be scheduled for executing jobs in $\mathcal{C}$. By definition, FR ensures that whenever k processors is used for a job with YS, at least k/m processors are used for that job in FR. Therefore, for the time when one processor is scheduled for J_0, at least $(s/m)w_1$ work is scheduled for $\mathcal{C}$. On the other hand, we know from ⟨3⟩ that the amount of work done on $\mathcal{C}$ by FR does not exceed that by YS (i.e. $w_0 + w_1$). Therefore,

$$(5.5) \quad s \sum_{j=\phi_0-1}^{m-1} \sum_{i=0}^{m-1} (m - j - \frac{i}{m}) x_{i,j} + \frac{s}{m} w_1 \le w_0 + w_1.$$

Together with the fact that $s \ge 2 - \frac{2}{m+1}$ and $w_0 \le \sum_{i=0}^{m-1} \sum_{j=\phi_0-1}^{m-1} x_i$, we obtain the following lower bound for $\sum_{j=\phi_0-1}^{m-1} x_{m,j}$ after eliminating w_0 and w_1 from (5.3)-(5.5).

$$\sum_{i=0}^{m-1} \sum_{j=\phi_0-1}^{m-1} \left(1 - \frac{si}{m} + \frac{2i(m-2-j)}{m^2+1-(m+1)\phi_0}\right) x_{i,j} \le (s-1) \sum_{j=\phi_0-1}^{m-1} x_{m,j}$$

This completes the inductive proof of (5.2), since the extra term on the left of the inequality above is never negative.

5.2 Trade-off between speed and processor In this section we sketch the effect when $m+p$ processors are available to the FR algorithm (when we know that the job set is feasible using only m processors). Note that the algorithm is still well defined. Although p extra processors are available to FR, we still use YS without any extra processor as the yardstick.

With more processors, FR achieves optimality with less speedy machines. More precisely, we show that FR is optimal when given $m+p$ speed-s processors, where $s \ge 2 - \frac{2(m-1)+mp}{(m+1)(m-1)+mp}$. The proof is similar to §5.1. In particular, we show Lemma 5.1 under this setting, by a contradiction between inequalities (5.1) and (5.2).

Note that ⟨1⟩-⟨3⟩ still hold. Since inequalities (5.1), (5.3) and (5.4) do not involve the amount of work done by FR for jobs other than J_0, these inequalities also hold under this setting.

To complete the inductive proof of (5.2), we improve (5.5) as follows. Again we study the work done on $\mathcal{C}$ with FR. Whenever FR schedules less than one processor for J_0, no processor can be idle. Since p more processors are available, this implies that there are more processors remaining which must be scheduled for executing jobs in $\mathcal{C}$. This results in the following inequality:

$$(5.6) \quad s \sum_{i=0}^{m-1} \sum_{j=\phi_0-1}^{m-1} (m + p - j - \frac{i}{m}) x_{i,j} + \frac{s}{m} w_1 \le w_0 + w_1.$$

Together with the fact that $s \ge \frac{2(m-1)+mp}{(m+1)(m-1)+mp}$ and the bound $w_0 \le \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} x_{i,j}$, we obtain from (5.3), (5.4) and (5.6) the following lower bound for $\sum_{j=\phi_0-1}^{m-1} x_{m,j}$.

$$\sum_{i=0}^{m-1} \sum_{j=\phi_0-1}^{m-1} \left(1 - \frac{si}{m} + T\right) x_{i,j} \le (s-1) \sum_{j=\phi_0-1}^{m-1} x_{m,j}$$

where

$$T = \frac{(s-1)(2m+p-2)}{(m-\phi_0-s+1)(m-1)^2(m+p-1)} (i((m-1)(m+p-2) - j(m+p-1)) + jmp)$$

Comparing with (5.2), we have an extra term T on the left of the inequality. T is never negative, completing the inductive proof of (5.2) and thus the proof of Lemma 5.1.

5.3 Number of migrations To bound the number of migrations, FR need to be more careful when it chooses a processor for each scheduled job. In the previous discussion, the scheduling of the processor P_l is completely determined. Jobs allocated to other processors are always allocated a full processor. We show that the following simple allocation strategy for processors other than P_l would guarantee that only $O(n)$ migrations are required, where n is the number of jobs released. After the algorithm is invoked and the set of jobs to be executed is determined, it compares the jobs to be executed in processors other than P_l before and after the invocation. For jobs which is allocated a processor both before and after the algorithm is invoked, it simply stays in the same processor. The remaining jobs are allocated arbitrarily in the remaining processors.

Now we bound the number of migrations required by the algorithm for jobs. Jobs in `reduced` mode always work on the same processor and thus need no migration. Each job may change to `reduced` mode only once, so in total we have at most n migrations when jobs change mode. For `full` mode jobs, FR always schedules the $m-1$ earliest deadline jobs which are in `full` mode in the processors other than P_l. Therefore the schedule in the first $m-1$ processors is exactly the same as an EDF schedule in which

- $m-1$ processors are available.
- The number of jobs, release times and deadlines of all jobs are exactly the same as that of the input for FR;
- The processing time of each job is the amount of work for the job which is scheduled to processors other than P_l in FR;

The number of migrations needed by this EDF schedule is $O(n)$. Therefore, in FR, jobs working in the first $m-1$ processors need only $O(n)$ migrations. Finally, `full` mode jobs may need to migrate to or from P_l, but there can only be one such migration per release and mode change, amounting to at most $2n$ migrations. Adding up all these, the number of migrations is $O(n)$.

6 Lower bound result for deadline-ordered algorithms

Both EDF and FR are deadline-ordered algorithms, with which the scheduling of jobs depends only on the relative order of job deadlines instead of their exact values. For such kind of algorithms, we obtain new lower bound results on the amount of extra speed and processors to achieve optimality:

THEOREM 6.1. *No deadline-ordered algorithm is optimal if only p extra processors are given and all processors are speed-s, where*

$$s < \frac{1}{1-(\frac{m-1}{m})^m + \frac{p}{m}(\frac{m-1}{m})^{m-1}}.$$

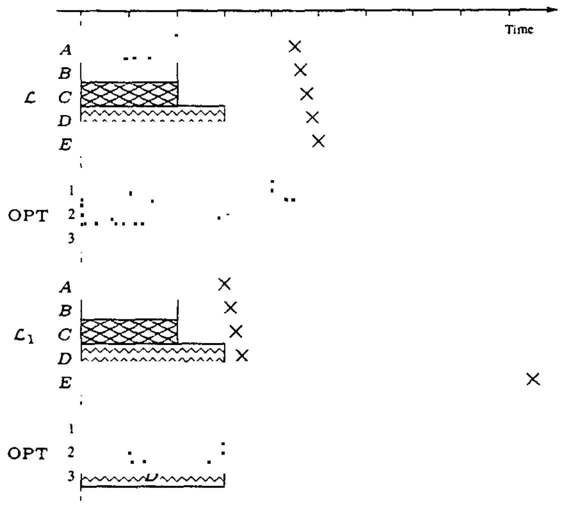

Figure 3: Lower bound example when $m=3$. The job lists $\mathcal{L}$ and $\mathcal{L}_1$ are illustrated, together with a possible schedule by an off-line algorithm.

Proof. Consider a collection of feasible job sets which are identical except the deadlines; the deadlines, though having different values, have the same relative ordering. The behavior of a deadline-ordered algorithm is the same for any of such job sets. Our lower bound argument exploits the this feature of deadline-ordered algorithms. We construct the following job set denoted $\mathcal{L}$, which is characterized by a small positive constant ϵ.

- One job for each $0 \le k \le m-1$; release time: 0, required work: 1, deadline $(m/(m-1))^{m-1} + k\epsilon$;
- One job for each $1 \le k \le m-1$; release time: 0, required work: $(m/(m-1))^k$, deadline $(m/(m-1))^{m-1} + (m-1+k)\epsilon$.

Since all jobs have the same release time and have deadlines before $(m/(m-1))^{m-1}$, an optimal off-line algorithm may simply schedule each job in turn, using up the period of time from 0 to $(m/(m-1))^{m-1}$ of a processor before considering the next processor. Since the total amount of work is $m + \sum_{k=1}^{m-1}(m/(m-1))^k = m(m/(m-1))^{m-1}$, m processors suffices to complete all jobs without missing any deadline.

For each $0 \le j \le m-1$, consider the following job set $\mathcal{L}_j$:

- One job for each $0 \le k \le m-1$; release time: 0, required work: 1, deadline $(m/(m-1))^j + k\epsilon$;

- One job for each $1 \leq k \leq j$; release time: 0, required work: $(m/(m-1))^k$, deadline $(m/(m-1))^j + (m-1+k)\epsilon$.
- One job for each $j+1 \leq k \leq m-1$; release time: 0, required work: $(m/(m-1))^k$, deadline $2(m/(m-1))^{m-1} + (m-1+k)\epsilon$.

By an argument similar to the feasibility of $\mathcal{L}$, the first $m+j$ jobs can all be completed by $(m/(m-1))^j$ by an off-line optimal algorithm. The remaining jobs can be scheduled exactly as $\mathcal{L}$ starting at time $(m/(m-1))^{m-1}$. Thus $\mathcal{L}_j$ is feasible. See Figure 3 for an example.

Note that the collection of two jobs $\{\mathcal{L}, \mathcal{L}_j\}$ satisfies the requirements set at the beginning of the proof, and thus are scheduled exactly the same by a deadline-ordered on-line algorithm. Since the algorithm is optimal, the first $m+j$ jobs must all be completed before the deadline of the $(m+j)$-th job in $\mathcal{L}_j$, i.e. before time $(m/(m-1))^j + (m-1+j)\epsilon$.

This implies that, during the execution of the deadline-ordered on-line algorithm for $\mathcal{L}$, only $m-j$ jobs remain in the system, and thus at least j processors must sit idle, after time $(m/(m-1))^j + (m-1+j)\epsilon$. By using a small enough ϵ, the sum of non-idle time of all processors can be made arbitrarily close to $p + (m-1) + m(m/(m-1))^{m-1}$. In these time the algorithm must completes all jobs, which total work is $m(m/(m-1))^{m-1}$. Dividing these two quantities gives the desired lower bound for the speed requirement of a deadline-ordered on-line algorithm. □

Putting $s = 1$ in Theorem 6.1, we obtain a lower bound for the number of speed-1 processors to achieve optimality.

COROLLARY 6.1. *Using speed-1 processors, a deadline-ordered algorithm needs at least $m-1$ extra processors to be optimal.*

References

[1] S. Albers, *Better bounds for online scheduling*, in Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, 1997, pp. 130–139.

[2] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan, *Flow and stretch metrics for scheduling continuous job streams*, in Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, 1998, pp. 270–279.

[3] P. Berman and C. Coulston, *Speed is more powerful than clairvoyance*, in Proceedings of the Sixth Scandinavian Workshop on Algorithm Theory, 1998. To appear.

[4] M. L. Dertouzos, *Control robotics: the procedural control of physical processes*, in Proceedings of IFIP Congress, 1974, pp. 807–813.

[5] M. L. Dertouzos and A. K. L. Mok, *Multiprocessor on-line scheduling of hard-real-time tasks*, IEEE Transactions on Software Engineering, 15 (1989), pp. 1497–1506.

[6] J. Edmonds, *Non-clairvoyant multiprocessor scheduling of jobs with arbitrary arrival times and changing execution characteristics*. Manuscript.

[7] J. Edmonds, D. D. Chinn, T. Brecht, and X. Deng, *Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics (extended abstract)*, in Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, 1997, pp. 120–129.

[8] M. R. Garey and D. S. Johnson, *Computers and intractability: a guide to the theory of NP-completeness*, W. H. Freeman, 1979.

[9] D. Gusfield, *Bounds for naive multiple machine scheduling with release times and deadlines*, Journal of Algorithms, 5 (1984), pp. 1–6.

[10] B. Kalyanasundaram and K. R. Pruhs, *Speed is as powerful as clairvoyance*, in Proceedings of the 36th Annual Symposium on Foundations of Computer Science, 1995, pp. 214–221.

[11] B. Kalyanasundaram and K. R. Pruhs, *Minimizing flow time nonclairvoyantly*, in Proceedings of the 38th Annual Symposium on Foundations of Computer Science, 1997, pp. 345–352.

[12] G. Koren, D. Shasha, and S. C. Huang, *MOCA: A multiprocessor on-line competitive algorithm for real-time system scheduling*, in Proceedings of the Fourteenth Real-Time Systems Symposium, 1993, pp. 172–181.

[13] C. A. Phillips, C. Stein, E. Torng, and J. Wein, *Optimal time-critical scheduling via resource augmentation*, in Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, 1997, pp. 140–149.