

Abstract of thesis entitled

**On the Effectiveness of Additional Resources
for On-line Firm Deadline Scheduling**

submitted by

Ngan Tsuen Wan

The performance of a real-time computer system is determined by the effectiveness of scheduling of jobs to meet their deadlines. However, many scheduling problems in the off-line setting have been proven to be NP-complete. In other words, it is computationally infeasible to find optimal schedules for many cases. The situation is further complicated by the fact that scheduling algorithms are on-line in nature, i.e., they do not have advance knowledge about the jobs until they are released. As expected, many scheduling problems do not admit on-line algorithms with optimal or reasonably good performance guarantee. A natural approach towards better performance guarantee is to allow on-line schedulers to have more resources (such as faster processors or extra processors). Intuitively, the additional resources compensate on-line schedulers for the lack of future information. Notice that most scheduling problems remain non-trivial even if a large amount of additional resources are available.

In this thesis, we revisit several classical deadline scheduling problems that aim at maximizing the total work or value of jobs that can be completed by their deadlines. We consider different settings, including uniprocessor and multiprocessor scheduling, and jobs with the same or different relative importance. We show new upper bounds and lower bounds on the effectiveness of using additional resources to provide better performance guarantee. Our results allow system administrators to compare and assess various scheduling algorithms and decide the resource requirement for their systems.

Note: *This is a slightly edited environmentally friendly version.*

Contents

1	Introduction	1
1.1	On-line Firm Deadline Scheduling	1
1.2	Additional Resources	2
1.3	Previous Work and Our Contributions	2
1.3.1	UFS-1 and UFS- k	2
1.3.2	MFS- k	3
1.4	Preliminaries	3
1.5	Organization	3
2	Scheduling with a Faster Processor	4
2.1	Lower Bound for UFS-1	4
2.2	Lower Bound for UFS- k	6
2.3	General Upper Bound for Tight Jobs	7
2.3.1	Algorithm	7
2.3.2	Basic Properties	8
2.3.3	Optimality	9
2.4	Concluding Remarks	11
3	Scheduling with Additional Processors	12
3.1	The EDF-Plus algorithm	12
3.2	Constant Competitiveness	13
3.3	Optimality	15
3.4	Concluding Remarks	17
4	Multiprocessor Scheduling	18
4.1	Competitiveness with Additional Processors	18
4.1.1	Lower bound	18
4.1.2	Upper bound	19
4.2	Additional Processors and Competitive Ratios	20
4.3	Concluding Remarks	22
5	Conclusion	23
5.1	Summary	23
5.2	Discussion	23
5.2.1	Speed versus processors	23
5.2.2	Optimality with a faster processor	23
5.2.3	Optimality in multiprocessor scheduling	23
5.3	Open Problems	23

1 Introduction

Thanks to the rapid development of science and technology, real-time computer systems now play a vital role in facilitating our daily lives. Examples include automated factories, traffic control systems, and stock exchange market. The performance of these real-time computer systems is determined by the effectiveness of the scheduling of jobs to meet their deadlines. However, many scheduling problems in the off-line setting have been proven to be NP-complete. In other words, it is computationally infeasible to find optimal schedules for many cases.

In reality, the situation is further complicated by the fact that scheduling algorithms are on-line in nature, i.e., they do not have advance knowledge about the jobs until they are released. As expected, many scheduling problems do not admit on-line algorithms with optimal or reasonably well performance guarantee. A natural approach towards better performance guarantee is to allow on-line schedulers to have more resources (such as faster processors or extra processors) [17, 23–26]. Intuitively, the additional resources compensate on-line schedulers for the lack of future information. Notice that most scheduling problems remain non-trivial even if large amount of additional resources are available. For example, the Earliest Deadline First algorithm (EDF) is known to be optimal for the underloaded single-processor deadline scheduling problem, yet for the overloaded case, neither EDF nor any algorithm can be optimal even arbitrary number of processors are allowed.

In this thesis we revisit several classical deadline scheduling problems, showing new upper bounds and lower bounds on the effectiveness of using additional resources to provide better performance guarantee. Our results allow system administrators to compare and assess various scheduling algorithms and decide the resource requirement for their systems.

1.1 On-line Firm Deadline Scheduling

The on-line firm deadline scheduling problem is defined as follows. There is a sequence of jobs to be scheduled for processing in $m \geq 1$ processors. Jobs are released in an unpredictable fashion. Every job is sequential in nature and can be processed by at most one processor at a time. Preemption is allowed at no cost. The processing time and deadline of a job are known only when the job is released. Each job is associated with a *value* (also known as *credit*), which reflects the importance of the job. The deadline is firm in the sense that the value of a job can be obtained only by completing it on or before its deadline. No value is obtained if the deadline is missed. A scheduling algorithm aims to maximize the total

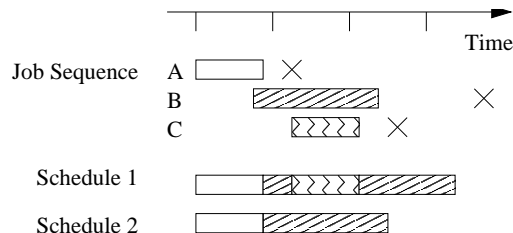


Figure 1: Two sample schedules. The job sequence comprises three jobs, each represented by a rectangle. Release time is denoted by the left edge. Required work is represented by the length. Deadline is denoted by a cross. Each schedule shows the time at which a job is executed. Note that Schedule 1 obtains the value of all the three jobs, while Schedule 2 only obtains the value of job A and B.

value of jobs that can be completed by their deadlines. See Figure 1 for an example.

A system is *underloaded* if there is a schedule meeting the deadline of every job released. In general, we do not have this guarantee, and a system may be *overloaded* and there is no schedule meeting every deadline. This model is applicable to systems where the required work exceeds the capacity and the scheduling algorithm has to decide which jobs to complete. It is motivated from embedded systems (see, e.g., [17, 21, 26] for discussion).

The *value density* of a job is equal to its value divided by its processing time. The *importance ratio* k of a system is the ratio of the largest possible value density to the smallest possible one. When $k = 1$, the value of a job is directly proportional to its processing time and the system is said to have *uniform value density*. For systems where some jobs are more important than the others, we have $k > 1$, and the system is said to have *general value density*. A scheduler in a uniform value density system aims at maximizing the processing time or the work of jobs that meet their deadlines.

We measure the performance of an on-line algorithm by comparing it with an *optimal off-line algorithm*, denoted OPT, on the same set of jobs. For any $c \geq 1$, an on-line algorithm is said to be *c-competitive* (or have a *competitive ratio* c) if, for any job sequence, it is guaranteed to achieve $1/c$ of the total value obtained by OPT. If the competitive ratio is a constant independent of both k and m , the on-line algorithm is said to be *competitive*. A 1-competitive algorithm is also said to be *optimal*. See [7, 27] for more background on competitive analysis.

For convenience, we denote UFS-1 as the on-line firm deadline scheduling problem on a uniprocessor system with uniform value density [3, 4, 23–26], and UFS- k as the on-

line firm deadline scheduling problem on a uniprocessor system with importance ratio k [3, 22, 25]. Similarly, we denote MFS-1 and MFS- k as on-line firm deadline multi-processor scheduling problem with importance ratio 1 and k [21, 25], respectively. In this thesis, we are going to study on-line algorithms for UFS-1, UFS- k , and MFS- k .

1.2 Additional Resources

Traditionally, the performance of an on-line algorithm is measured by comparing it with the optimal off-line algorithm in the worst case. Yet this may not be a good measurement for many deadline scheduling problems. Indeed, some early lower bound results [3,4] suggested that all on-line algorithms for some deadline scheduling problems perform equally bad. The traditional competitive analysis has failed to differentiate good and bad algorithms.

In recent years, there is a new approach for studying better performance guarantee without making assumptions on future inputs; the basic idea is to allow the on-line algorithm to have more resources than the off-line algorithm (e.g. see [17, 23–26]). Intuitively, the additional resources are needed to compensate the on-line algorithm for the lack of future information. The key question is whether a moderate amount of additional resources can provide satisfactory competitiveness or even attain optimality. This kind of analysis assesses the amount of additional resources needed to meet the optimal or competitive requirement of a system. This is also known as *resource augmentation analysis* in the literature.

To ease our discussion of the comparison between on-line algorithms using additional resources and the optimal offline algorithm, denoted OPT, we use the following notations:

Definition.

- A scheduling algorithm \mathcal{A} is said to be speed- s c -competitive if \mathcal{A} which uses speed- s processors can guarantee to obtain a fraction $1/c$ of the total value obtained by OPT, which uses speed-1 processors. A processor is said to be speed- s if it can process 1 unit of work in $1/s$ time, i.e., s times faster than a processor used by OPT.
- A scheduling algorithm \mathcal{A} is said to be w -processor c -competitive if \mathcal{A} , whose number of processors is w times more than OPT, can guarantee to obtain a fraction $1/c$ of the total value obtained by OPT.
- A scheduling algorithm \mathcal{A} is said to be speed- s w -processor c -competitive if \mathcal{A} , whose number of processors is w times more than OPT and all processors are speed- s , can guarantee to obtain a fraction $1/c$ of the total value obtained by OPT.

- A w -processor speed- s optimal algorithm refers to a w -processor speed- s 1-competitive algorithm.

1.3 Previous Work and Our Contributions

In this thesis, we study the problems UFS-1, UFS- k , and MFS- k . We consider on-line algorithms with additional resources, namely, faster and/or additional processors, for attaining a constant competitive ratio or even optimality. We give new upper and lower bound results on the effectiveness of additional resources.

1.3.1 UFS-1 and UFS- k

Without using additional resources, no optimal scheduling algorithm can exist for either UFS-1 or UFS- k . Note that optimality is desirable, especially for underloaded systems where the optimal off-line algorithm can meet all deadlines. For UFS-1, Baruah et al. [4] showed a lower bound of 4 on the competitive ratio. Baruah et al. [3] presented a 4-competitive algorithm called TD_1 . In the same paper, they also showed a lower bound of $(1 + \sqrt{k})^2$ on the competitive ratio for UFS- k . Subsequently, Koren and Shasha [22] gave an algorithm called D^{over} with a matching upper bound.

A Faster Processor. For UFS-1 and particularly UFS- k , when a faster processor is allowed, Kalyanasundaram and Pruhs [17] gave an algorithm called SLACKER that can improve the competitive ratio to a constant, i.e., independent of k . Precisely, for any real $\delta > 0$, SLACKER is speed- $(1 + 2\delta)$ λ -competitive, where $\lambda = (1 + \delta^{-1})(1 + \delta^{-1/2})(1 + \delta^{-1/2} + \delta^{-1})$. For example, putting $\delta = 1/2$, SLACKER is speed-2 32-competitive. The only algorithm that can make use of a faster processor to obtain optimal scheduling is by Lam and To [24]. Precisely, they showed that an algorithm called EDF-AC is speed-2 optimal for UFS-1 and speed-4 $\lceil \log k \rceil$ optimal for UFS- k . No non-trivial lower bound greater than 1 on the speed requirement for attaining optimality is known for either UFS-1 and UFS- k .

Our first contribution is two lower bound results: we show that there is no speed- s optimal algorithm for UFS-1 when $s < \phi$, where ϕ is the golden ratio (approximately 1.618). For UFS- k , the lower bound can be improved to 2. Motivated by the gap between the lower bound of 2 and upper bound of $4 \lceil \log k \rceil$ for UFS- k , we attempt to devise algorithms with a lower speed requirement. We believe that jobs with tight deadlines, i.e., the deadline of a job is equal to its release time plus processing time, are the most difficult to handle. We give a speed- $O(1)$ optimal algorithm when all jobs are tight. Our

result serves as a first step to finding a speed- $O(1)$ optimal algorithm for the general case.

Additional Processors. Less result is known for using additional processors. Indeed, using additional processors is harder than using faster processors; for instance, a speed-2 processor can simulate two speed-1 processors by time-sharing, but the reverse is not true (as jobs are sequential in nature). The only previous result that can exploit additional processors is for UFS-1, where Baruah [2] gave a m -processor $m/(m-1)$ -competitive algorithm. No optimal algorithm based on additional processors has been known for either UFS-1 or UFS- k .

We present the first result on attaining constant competitive ratio and optimality using additional processors. Precisely, we give a 2-processor optimal for UFS-1. Based on this result, we show a $2 \lceil \log k \rceil$ -processor 2-competitive algorithm for UFS- k . In addition, we present a 4-processor optimal algorithm for UFS-1, which can also be extended to give a $4 \lceil \log k \rceil$ -processor optimal algorithm for UFS- k . More interestingly, we also show that no w -processor algorithm is c -competitive for UFS- k for any constant c unless $w = \Omega(\log k)$.

Based on the above results, we can compare the power of faster processors with additional processors. For UFS- k , we have a speed- $O(1)$ optimal algorithm for the difficult case when all jobs are tight, yet we also have an $\Omega(\log k)$ lower bound on the extra processor requirement even for attaining a competitive ratio independent of k . Thus, in terms of additional resource requirement, using faster processors is a more cost effective way than extra processors.

1.3.2 MFS- k

Without using additional resources, Korea and Shasha [21] decided an algorithm MOCA with competitive ratio $1 + m(k^{1/\psi} - 1)$, where $\psi = \frac{m}{2} \frac{\log k}{\log k + 1}$. They also gave a lower bound of $\frac{k}{k-1} m(k^{1/m} - 1)$ on the competitive ratio. These bounds tend to $O(\log k)$ when m tends to infinity.

When faster processors are allowed, Lam and To [25] extended the SLACKER algorithm for MFS- k and improved the algorithm to speed- $(1 + 2\delta) (1 + 2\delta^{-1} + 4\delta^{-2})$ -competitive. For instance, when $\delta = 1/2$, the competitive ratio is improved from 32 to 21.

To exploit additional processors for MFS- k , we extend MOCA to an $O(\log k)$ -processor $O(1)$ -competitive algorithm. We also show that $\Omega(\log k)$ times extra processors are required to achieve a constant competitive ratio.

Besides, we consider an extension of MSLACKER. Precisely, the competitive ratio of MSLACKER can be improved using additional faster processors instead of even faster processors. For example, MSLACKER can be improved from speed-2 21-competitive to speed-3 7-competitive by further increasing the processor speed, or it can be improved to 5-processor speed-2 5-competitive by using extra speed-2 processors.

1.4 Preliminaries

For any job J , let $r(J)$, $p(J)$, $d(J)$, and $v(J)$ denote the release time, processing time, deadline, and value of J , respectively. The value density $\rho(J) = v(J)/p(J)$. For convenience, we normalize the smallest value density to be 1, and we assume all jobs have value density in the range $[1, k]$. The *span* of J refers to the interval $[r(J), d(J)]$. Since completing a job after its deadline gives no credit, whenever we say a job is completed, it is meant that the job is completed by its deadline. A processor is said to be *idle* if it is not running a job, and *busy* otherwise. Similarly, a job is said to be *idle* if it is not running on a processor, and *busy* otherwise.

An on-line scheduling algorithm operates as follows. 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.

1.5 Organization

The remainder of the thesis is organized as follows. Chapter 2 presents upper and lower bound results for UFS-1 and UFS- k using a faster processor. Chapter 3 shows upper and lower bound results for UFS-1 and UFS- k using more than one speed-1 processor. In chapter 4, we give competitive algorithms for MFS- k using additional and/or faster processors. Finally, we summarize the results and discuss some possible future directions in chapter 5.

¹Unless otherwise specified, all logarithms are base 2 in this thesis.

2 Scheduling with a Faster Processor

In this chapter, we study algorithms using a faster processor for the problems UFS-1 and UFS- k . Precisely, we study the performance of on-line algorithms using one speed- s processor, where $s > 1$, as compared with an off-line algorithm using one speed-1 processor.

Lower bound results. Lam and To [25] showed that EDF-AC is speed-2 optimal for UFS-1 and speed-4 $\lceil \log k \rceil$ optimal for UFS- k . Yet no lower bound on the speed requirement for achieving optimality has been known for either problem. In section 2.1, we give the first non-trivial lower bound for UFS-1. Specifically, we show that no algorithm is speed- s optimal for UFS-1 unless $s \geq \phi$, where ϕ is the golden ratio. In section 2.2, we show that the lower bound for UFS- k can be improved slightly to 2.

Upper bound result. Motivated by the gap between the upper bound of $4 \lceil \log k \rceil$ and the lower bound of 2 for UFS- k , we turn our attention to finding an algorithm for UFS- k that can reduce the speed requirement for optimality. We believe that jobs with tight deadlines are the most difficult to handle. In fact, existing lower bounds on competitive ratios when no additional resource is available for UFS-1 and UFS- k [3, 4], and our lower bound on speed requirement for UFS-1 are all based on the tight deadline setting. When jobs are not tight, only a weaker lower bound on the competitive ratio for UFS-1 is known [8]. Based on this observation, we study algorithms for handling tight jobs. In section 2.3, we give a new algorithm that is speed- $O(1)$ optimal for UFS- k when all jobs are tight. Note that the speed factor is independent of k . We believe that scheduling jobs with tight deadlines is no easier than the general problem, and our result serves as a first step to finding a speed- $O(1)$ optimal algorithm for the general case.

2.1 Lower Bound for UFS-1

Recall that EDF-AC is a speed-2 optimal algorithm for UFS-1. This section shows that for UFS-1, the speed requirement for any optimal algorithm is at least the golden ratio (denoted ϕ), which is the solution of the equation $\phi^2 = \phi + 1$. I.e., $\phi = (1 + \sqrt{5})/2 \approx 1.618$.

The following lemma states a property of ϕ that our lower bound argument makes use of. Basically, it implies that a speed- s processor, where $s < \phi$, cannot complete $s + 1$ units of work in s units of time.

Lemma 2.1. *For $1 \leq s < \phi$, $1 + \frac{1}{s} > s$.*

Proof. $1 + \frac{1}{s} > 1 + \frac{1}{\phi} = \phi > s$. □

To simplify the proof, we show that it suffices to consider on-line algorithms that are *busy* in the following sense.

Definition. At any time t during the span of a job J , we say that J is *feasible* with respect to a speed- s processor if J is not yet completed at time t and it is still possible schedule J to meet the deadline. I.e., the remaining work of J is at most $s(d(J) - t)$.

Definition. A scheduling algorithm \mathcal{A} is a *busy* algorithm if it schedules a feasible job (with respect to the processors used by \mathcal{A}) whenever such a job is available.

Lemma 2.2. *For any scheduling algorithm \mathcal{A} , there exists a busy scheduling algorithm \mathcal{A}' (using a processor of the same speed) such that \mathcal{A}' can meet the deadlines of all the jobs that \mathcal{A} can meet.*

Proof. Given an algorithm \mathcal{A} , we construct \mathcal{A}' as follows. At any time t , \mathcal{A}' , based on a simulation of \mathcal{A} , determines which job is currently scheduled by \mathcal{A} . Denote \mathcal{F}_t as the set of feasible jobs in \mathcal{A}' at time t . If \mathcal{A} is scheduling a job J in \mathcal{F}_t , \mathcal{A}' also schedules J . Otherwise, \mathcal{A}' arbitrarily chooses a job in \mathcal{F}_t if $\mathcal{F}_t \neq \emptyset$.

Consider any job J completed by \mathcal{A} . Below we show that at any time, the work done on J by \mathcal{A}' is always at least as much as that of \mathcal{A} , which implies that \mathcal{A}' also completes J . Lemma 2.2 thus follows. Suppose, for the sake of contradiction, that t is the first time instant when the work done on J by \mathcal{A} exceeds that of \mathcal{A}' . This means that \mathcal{A} schedules J at time t but \mathcal{A}' does not. Since J is finally completed by \mathcal{A} , J is feasible for \mathcal{A} at t . As the work done by \mathcal{A} and \mathcal{A}' are the same just before t , J is also feasible for \mathcal{A}' at t . By the definition of \mathcal{A}' , \mathcal{A}' should schedule J at t . This leads to a contradiction. □

We then prove the following theorem that leads to the lower bound result.

Theorem 2.3. *There is no busy algorithm that is speed- s optimal for UFS-1 unless $s \geq \phi$.*

The proof of this main theorem is quite involved, though the basic idea is simple. We consider the off-line algorithm as an adversary who generates the input job sequence while looking at the response from the on-line algorithm and exhibits its schedule after the entire input has been generated. The adversary's goal is to maximize the total value it obtains while keeping it down for the on-line algorithm.

Initially, \mathcal{A} is given two jobs such that \mathcal{A} cannot complete both of them. To be optimal, \mathcal{A} must complete the longer job. When the shorter job becomes not feasible, the third job, which is even longer, is released. Again, \mathcal{A} cannot complete

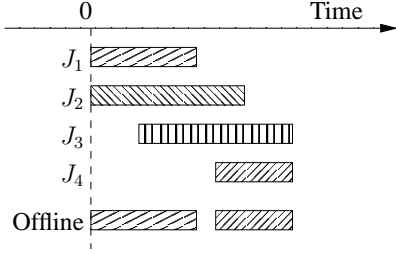


Figure 2: The input job sequence and the optimal off-line schedule

both current jobs and is forced to switch to the third one without completing the first two jobs. This process ensures that the adversary can indeed complete more than one job. Details are as follows.

Let \mathcal{A} be any busy on-line algorithm that is speed- s optimal, where $s < \phi$. To show that \mathcal{A} is not optimal, we consider a sequence of four jobs (J_1, J_2, J_3 , and J_4). All the jobs are tight, i.e., $d(J) = r(J) + p(J)$ for any job J . J_1 and J_2 are both released at time 0 with processing times 1 and s , respectively. The release time of J_3 and J_4 depend on how \mathcal{A} schedules J_1 and J_2 . The argument below makes use of two parameters depending on s . Let $\delta = 1 - s + 1/s$, and let $\lambda = \delta/\phi$. By Lemma 2.1, δ , as well as λ , is greater than 0.

The time required for a speed- s processor to complete both J_1 and J_2 is $\frac{s+1}{s} > s$. Recall that $d(J_1) = 1$ and $d(J_2) = s$. Thus, \mathcal{A} cannot complete both jobs. To be optimal, \mathcal{A} must abandon J_1 and complete J_2 , as the latter has a longer processing time. By definition, J_1 is feasible at time 0 and not feasible at time 1. Let t be the last instant such that J_1 is still feasible, and let ℓ be the total processing time J_1 received up to time t .

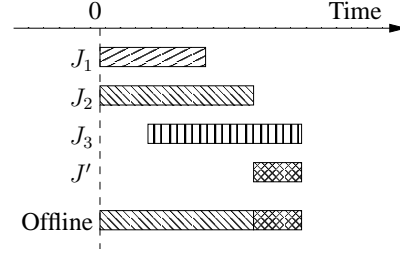
Fact 2.4. $s(1 - t) + s\ell = p(J_1) = 1$, or equivalently, $t = 1 + \ell - 1/s$.

Intuitively, J_3 is released soon after J_1 becomes not feasible. Let $r(J_3) = t + \lambda = 1 + \ell - 1/s + \lambda$, and let $p(J_3) = s(1 + \lambda)$.

Claim 2.5. *The work left for J_2 at time $r(J_3)$ is $1 - s\lambda$.*

Proof. Since \mathcal{A} is a busy scheduling algorithm, the processing time received by J_2 up to time $r(J_3)$ is the processing time not received by J_1 . By the property of busy scheduling algorithm, \mathcal{A} does not schedule J_1 after t . Thus, at time $r(J_3)$, the work done on J_1 is still $s\ell$; and for J_2 , the work done is $s(r(J_3) - \ell) = s - 1 + s\lambda$ and the remaining work is $s - (s - 1 + s\lambda) = 1 - s\lambda$. \square

Next, we observe that \mathcal{A} , using a speed- s processor, cannot complete both J_2 and J_3 on or before $\max(d(J_2), d(J_3))$,


 Figure 3: The input job sequence to bound ℓ'

which is $s(1 + \lambda)$ units of time after t . It is simply because the total work required is too much. See the following claim.

Claim 2.6. *A speed- s processor cannot process the remaining work of J_2 and J_3 within $s(1 + \lambda)$ units of time.*

Proof. The speed required to complete both jobs is at least

$$\begin{aligned} & \frac{1 - s\lambda + s(1 + \lambda)}{s(1 + \lambda)} \\ &= \frac{1 + s}{s + s\lambda} \\ &= \frac{s^2 + s\delta}{s(1 + \lambda)} \quad (\text{by definition of } \delta) \\ &= \frac{s^2 + s\phi\lambda}{s(1 + \lambda)} \quad (\text{by definition of } \lambda) \\ &> \frac{s^2 + s^2\lambda}{s(1 + \lambda)} \quad (\text{because } \phi > s) \\ &= s. \quad \square \end{aligned}$$

Since $p(J_3) > p(J_2)$, \mathcal{A} must abandon J_2 and complete J_3 so as to guarantee optimality. Let t' be the last instant that J_2 is still feasible. Let ℓ' be the processing time received by J_2 during $[r(J_3), t']$. Notice that ℓ' cannot be too large as J_2 is finally abandoned. See the following claim for details.

Claim 2.7. $\ell' \leq 1/s^2 - \lambda$.

Proof. Suppose, for the sake of contradiction, that $\ell' > 1/s^2 - \lambda$. Consider the scenario that after $d(J_2)$, a tight job J' is released at time s with the same deadline as J_3 . I.e., $p(J') = d(J_3) - s = 1 + \ell + \lambda + s\lambda - 1/s$. The adversary can complete J_2 and J' . Therefore, \mathcal{A} , after abandoning J_1 and J_2 , must complete both J_3 and J' in order to be optimal. The speed required is at least

$$\begin{aligned} & \frac{(s + s\lambda) + s\ell' + (1 + \ell + \lambda + s\lambda - 1/s)}{s + s\lambda} \\ & \geq 1 + \frac{s\ell' + 1 + \lambda + s\lambda - 1/s}{s(1 + \lambda)} \end{aligned}$$

$$\begin{aligned}
 &> 1 + \frac{(1/s - s\lambda) + 1 + \lambda + s\lambda - 1/s}{s(1 + \lambda)} &= 1 + \frac{s(1 - 1/s)\delta + 1/s + s\ell}{1 + \ell + (s - 1)\lambda} \\
 &= 1 + \frac{1 + \lambda}{s(1 + \lambda)} &= 1 + \frac{1}{s} \cdot \frac{1 + s^2\ell + s(s - 1)\delta}{1 + \ell + (s - 1)\lambda} \\
 &= 1 + 1/s &> 1 + 1/s \quad (\text{because } s \geq 1 \text{ and } \delta \geq \lambda) \\
 &> s \cdot \quad (\text{by Lemma 2.1}) &> s \cdot \quad \square
 \end{aligned}$$

Thus, \mathcal{A} can only complete either J_3 or J' and \mathcal{A} is not optimal. A contradiction occurs. \square

For the rest of the proof, we show that by adding a final job J_4 , \mathcal{A} can only complete either J_3 or J_4 , yet the adversary can complete both J_1 and J_4 , completing more work than \mathcal{A} no matter which job \mathcal{A} chooses.

J_4 is released after J_2 becomes not feasible. By the definition of t' , we have $s(s - t') + s\ell' = 1 - s\lambda$, or equivalently, $t' = s + \ell' - 1/s + \lambda$. Let $r(J_4) = t' + \lambda$, and let $p(J_4) = r(J_3) - r(J_4) + p(J_3) = 1 + \ell - \ell' - \lambda + s\lambda$.

Claim 2.8. *The work left for J_3 at time $r(J_4)$ is $2s - s^2 + s\ell$.*

Proof. As $r(J_4) > t'$, J_2 is not feasible at time $r(J_4)$. Since \mathcal{A} is a busy scheduling algorithm, the processing time received by J_3 during the interval $[r(J_3), r(J_4)]$ is the processing time not received by J_2 . During $[r(J_3), r(J_4)]$, the work done on J_2 is $s\ell'$, and the work done on J_3 is $s(r(J_4) - r(J_3) - \ell') = s^2 - s\ell - s + s\lambda$. So the work left for J_3 at time $r(J_4)$ is $s(1 + \lambda) - (s^2 - s\ell - s + s\lambda) = 2s - s^2 + s\ell$. \square

The following claim shows that \mathcal{A} cannot complete both J_3 and J_4 before their deadlines.

Claim 2.9. *A speed- s processor cannot process the remaining work of J_3 and J_4 within $s(1 + \lambda)$ units of time.*

Proof. The speed required for completing both J_3 and J_4 is at least

$$\begin{aligned}
 &\frac{p(J_4) + (2s - s^2 + s\ell)}{p(J_4)} \\
 &= 1 + \frac{2s - s^2 + s\ell}{p(J_4)} \\
 &= 1 + \frac{2s - s^2 + s\ell}{1 + \ell - \ell' + (s - 1)\lambda} \\
 &\geq 1 + \frac{2s - s^2 + s\ell}{1 + \ell + (s - 1)\lambda} \\
 &= 1 + \frac{s(2 - s - 1/s^2) + 1/s + s\ell}{1 + \ell + (s - 1)\lambda} \\
 &= 1 + \frac{s(1 - 1/s)(1 - s + 1/s) + 1/s + s\ell}{1 + \ell + (s - 1)\lambda}
 \end{aligned}$$

Recall that \mathcal{A} does not complete J_1 and J_2 . Together with the fact that \mathcal{A} cannot complete both J_3 and J_4 , we know that the work that can be completed by \mathcal{A} is at most $\max(p(J_3), p(J_4)) = p(J_3)$.

Next, we show that $r(J_4) > d(J_1) = 1$; thus, the adversary (using a speed-1 processor) can complete both J_1 and J_4 .

$$\begin{aligned}
 r(J_4) &= s + \ell' + 2\lambda - 1/s \\
 &\geq 2\lambda + s - 1/s \\
 &> (1 - s + 1/s) + s - 1/s \\
 &\quad (\text{because } \lambda = \delta/\phi > \delta/2) \\
 &= 1
 \end{aligned}$$

Finally, we show that $p(J_1) + p(J_4) > p(J_3)$. Thus, the adversary can obtain more work done than \mathcal{A} . In other words, \mathcal{A} is not optimal. A contradiction occurs.

$$\begin{aligned}
 &p(J_1) + p(J_4) - p(J_3) \\
 &= 1 + r(J_3) - r(J_4) \quad (\text{by definition of } p(J_4)) \\
 &= 1 + (1 + \ell - 1/s + \lambda) - (s + \ell' - 1/s + 2\lambda) \\
 &= 2 - s + \ell - \ell' - \lambda \\
 &\geq 2 - s - \ell' - \lambda \\
 &\geq 2 - s - 1/s^2 \quad (\text{by Claim 2.7}) \\
 &= (1 - 1/s)(1 + 1/s - s) \\
 &> 0
 \end{aligned}$$

We have completed the proof of Theorem 2.3. Based on Lemma 2.2, we can extend the lower bound result to any online algorithm that is not busy.

Corollary 2.10. *There is no speed- s optimal algorithm for UFS-1 unless $s \geq \phi$.*

2.2 Lower Bound for UFS- k

In this section we give a lower bound for UFS- k with importance ratio $k > 1$.

Lemma 2.11. *There is no speed- s optimal algorithm for UFS- k unless $s \geq 2$.*

Suppose, for the sake of contradiction, that there exists a speed- s optimal algorithm, where $s = 2 - \varepsilon$. Let $n = \lceil 1/\varepsilon \rceil$. Considering the input job sequence shown in Table 1. Jobs are released in pairs (J_i, J'_i) . The input terminates either when \mathcal{A} misses any deadline, or after the final job J_{n+1} is released at time $n + 1$. Notice that the importance ratio of the job sequence is $2^n = 2^{\lceil \log k \rceil} \leq k$.

Before the job pair (J_1, J'_1) is released at time 1, the system is underloaded. To be optimal, \mathcal{A} must complete J_0 at time 1. In general, \mathcal{A} must meet every deadline. See the following claim for details.

Claim 2.12. *\mathcal{A} must meet the deadlines of the jobs J'_{i-1} and J_i for $1 \leq i \leq n$.*

Proof. Notice that the two jobs have the same deadline $(i + 1)$ and value (2^{i-1}) . Suppose, for the sake of contradiction, that one of them missed its deadline at time $i + 1$. The input is stopped immediately (i.e., the last job pair is J_i and J'_i). The adversary completes J'_0, J'_1, \dots, J'_i as well as J_i , obtaining a total value of

$$(1 + 2 + 4 + \dots + 2^i) + 2^{i-1} = 2^{i+1} + 2^{i-1} - 1 .$$

The best \mathcal{A} can do is to complete all jobs except J'_{i-1} or J_i , obtaining a total value of

$$\begin{aligned} & (1 - \varepsilon) + 2 \times (1 + 2 + \dots + 2^{i-2}) + 2^{i-1} + 2^i \\ &= (1 - \varepsilon) + 2^i - 2 + 2^{i-1} + 2^i \\ &= 2^{i+1} + 2^{i-1} - 1 - \varepsilon , \end{aligned}$$

which is less than that of the adversary. This contradicts to the assumption. \square

Similarly, \mathcal{A} must complete the last two jobs J'_{i-1} and J_i to match the total value of the adversary. The total processing time of all jobs is $(1 - \varepsilon) + 1 + 2n + 1 = 2n + 3 - \varepsilon$. \mathcal{A} , using a speed- $(2 - \varepsilon)$ processor, can complete all the jobs no earlier than

$$\frac{2n + 3 - \varepsilon}{2 - \varepsilon} = n + \frac{n\varepsilon + 3 - \varepsilon}{2 - \varepsilon} \geq n + \frac{4 - \varepsilon}{2 - \varepsilon} > n + 2 ,$$

which is after the deadline of J_{n+1} . Thus, \mathcal{A} missed the deadline of some job. This leads to a contradiction and completes the proof of Lemma 2.11.

2.3 General Upper Bound for Tight Jobs

The best known result for UFS- k requires a speed- $4 \lceil \log k \rceil$ processor [25]. Note that when k is big, demanding a processor that is $4 \lceil \log k \rceil$ times faster may not be practical. A

natural question is whether the speed requirement for optimality in the case of general k can be improved to $o(\log k)$ or even $O(1)$.

In this section, we address this problem with a focus on tight jobs. Recall that a tight job has its deadline equals to its release time plus processing time. We present a two-processor speed- s optimal algorithm for UFS- k when all jobs are tight. Since two speed- s processors can be simulated by one speed- $2s$ processor, this algorithm can be considered as a speed- $2s$ optimal algorithm.

2.3.1 Algorithm

Definition. Let $r = \frac{1}{2} + \frac{1}{2s} \leq 1$. A job J , once released, is said to be *fresh* up to the time $d(J) - r \cdot p(J)$. (Roughly speaking, a job is no longer fresh when the time is close to its deadline.)

The following lemma states an important property of a fresh job.

Lemma 2.13. *While a job J is fresh, it is feasible to complete J on or before its deadline using a speed- s processor.*

Proof. Consider any time t when a job J is fresh. By definition, $d(J) - t \geq r \cdot p(J) = (\frac{1}{2} + \frac{1}{2s})p(J)$. Note that a speed- s processor needs $p(J)/s$ time to complete J . Since $s \geq 1$, we have $\frac{1}{2} \geq \frac{1}{2s}$, or equivalently, $\frac{1}{2} + \frac{1}{2s} \geq \frac{1}{s}$. Thus, starting from time t , a speed- s processor must be able to complete J on or before its deadline. \square

Algorithm 2.1 gives the details on the new on-line scheduling algorithm. The algorithm maintains a pool P of jobs that will be given priority for scheduling. Intuitively, if a job is still fresh and has sufficiently large value density, the algorithm will put it into the pool for possible scheduling. Among the jobs in the pool, the algorithm always schedules the two most dense jobs available. Once a job is scheduled, it can be preempted by a newly released job with sufficiently high density. Note that there is no guarantee that a job, once scheduled, will complete.

In the following, we will show that for scheduling jobs with tight deadlines², this algorithm, when given a speed-14 processor (or two speed-7 processors), is optimal.

Consider a job J that the on-line algorithm fails to meet its deadline, but an off-line algorithm can meet its deadline. During the span of J , the on-line algorithm must process other jobs for a considerably long period. These jobs should have reasonably high value density, yet the on-line algorithm may not complete them at the end and generates any value. The

²Recall that a deadline is tight if it is equal to the release time plus the processing time of the job.

Job	Release time	Processing time	Value density	Deadline
J_0	0	$1 - \varepsilon$	1	1
J'_0	0	1	1	2
J_1	1	1	1	2
J'_1	1	1	2	3
		\vdots		
J_i	i	1	2^{i-1}	$i + 1$
J'_i	i	1	2^i	$i + 2$
		\vdots		
J_n	n	1	2^{n-1}	$n + 1$
J'_n	n	1	2^n	$n + 2$
J_{n+1}	$n + 1$	1	2^n	$n + 2$

Table 1: The input job sequence for general value density

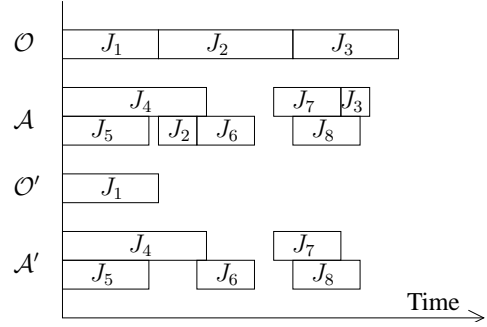
(1)	Initialization:
(2)	$P \leftarrow \emptyset$
(3)	
(4)	When Job J is released:
(5)	if $ P < 2$ or $\rho(J) \geq 2 \max_{T \in P} \rho(T)$
(6)	$P \leftarrow P \cup \{J\}$
(7)	Schedule the two most dense jobs, if available, in P
(8)	
(9)	When Job J completes:
(10)	$P \leftarrow P - \{J\}$
(11)	if there exists a fresh job not in P
(12)	Denote J_0 as the most dense fresh job not in P
(13)	if $ P < 2$ or $\rho(J_0) \geq 2 \max_{T \in P} \rho(T)$
(14)	$P \leftarrow P \cup \{J_0\}$
(15)	Schedule the two most dense jobs, if available, in P
(16)	
(17)	When Job $J \in P$ is certain to miss its deadline:
(18)	$P \leftarrow P - \{J\}$ // Discard J

Algorithm 2.1: A new algorithm for tight jobs

optimality of the algorithm is proven by a non-trivial amortization scheme, showing that the on-line algorithm will complete some extra jobs (comparing with an off-line algorithm) within or beyond the span of J , which can be used to pay off the value of J .

2.3.2 Basic Properties

Let \mathcal{A} and \mathcal{O} denote the schedules produced by our algorithm (using two speed- s processors) and an off-line algorithm (using one speed-1 processor) for any particular job set. Denote $\mathcal{J}(\mathcal{A})$ as the set of jobs that has ever been scheduled


 Figure 4: A sample schedule for \mathcal{O} and \mathcal{A} and the respective \mathcal{O}' and \mathcal{A}' . Note that J_2 and J_3 are removed in schedule \mathcal{O}' and \mathcal{A}' .

by \mathcal{A} , and similarly for $\mathcal{J}(\mathcal{O})$. Without loss of generality, we assume that every job that has ever been scheduled by \mathcal{O} can be completed by its deadline. Note that such an assumption is not valid for \mathcal{A} .

We construct schedules \mathcal{A}' and \mathcal{O}' from \mathcal{A} and \mathcal{O} respectively by removing all jobs that are completed by both \mathcal{A} and \mathcal{O} . An example is shown in Figure 4. By definition, for every job $J \in \mathcal{J}(\mathcal{O}')$, both \mathcal{O} and \mathcal{O}' schedule J to completion by its deadline, but neither \mathcal{A} nor \mathcal{A}' can complete J by its deadline.

Lemma 2.14. *For any two jobs scheduled by \mathcal{O} , their span must be disjoint.*

Proof. Recall that any job scheduled by \mathcal{O} can be completed by its deadlines. Since all jobs have tight deadlines, \mathcal{O} or any off-line algorithm can complete a job only by scheduling it throughout its entire span. Thus, any two jobs that can be

completed by \mathcal{O} must have non-overlapping span. \square

Lemma 2.15. *If \mathcal{A} misses the deadline of job J , then at any time while J is fresh, \mathcal{A} schedules at least one job with value density greater than $\rho(J)/2$.*

Proof. Let J be a job for which \mathcal{A} misses the deadline. Consider any time t when J is fresh. If J is in P , \mathcal{A} is executing some job(s) of value density at least $\rho(J)$. It remains to consider the case where J is not in P at time t . In this case, J is not in P during the entire period $[r(J), t]$ (otherwise, by Lemma 2.13, even at time t , it is still feasible for \mathcal{A} to complete J by its deadline and J , once put into P , will not be removed up to time t). Thus, \mathcal{A} must be executing some job(s) of value density greater than $\rho(J)/2$ during the entire period $[r(J), t]$. In either case, \mathcal{A} is executing a job with value density greater than $\rho(J)/2$. \square

Lemma 2.16. *At any time, let J_i and J_{i+1} be the i -th and $(i+1)$ -st most dense jobs in P . If J_{i+1} is not the least dense job in P , then $\rho(J_i) \geq 2\rho(J_{i+1})$.*

Proof. When a job J is added to P , either there was at most one job in P , or the density of J is at least double of the most dense job in P . Removing a job from P does not invalidate this property. \square

2.3.3 Optimality

Denote $\|\mathcal{A}'\|$ and $\|\mathcal{O}'\|$ as the total value of the jobs completed by schedules \mathcal{A}' and \mathcal{O}' , respectively. This subsection shows that if $s \geq 7$, $\|\mathcal{A}'\| \geq \|\mathcal{O}'\|$, or equivalently, $\|\mathcal{A}\| \geq \|\mathcal{O}\|$. It follows that the algorithm shown, when given two speed-7 processors or a speed-14 processor, can always match the off-line algorithm on the total value obtained.

Consider the schedule \mathcal{A}' . For any job $J \in \mathcal{J}(\mathcal{A}')$, let $T(J)$ be the total time \mathcal{A}' schedules J . At any particular time, J is said to be the primary job scheduled by \mathcal{A}' if \mathcal{A}' schedules J at that time and either there is no other job scheduled at the same time, or J has a higher job density than the other job scheduled (we break the tie by requiring J to be the job with smaller job identity). For a job J_0 scheduled by \mathcal{O}' , let $T_{J_0}(J)$ be the total time \mathcal{A}' schedules J as the primary job during the period of time when J_0 is fresh.

To prove that $\|\mathcal{A}'\| \geq \|\mathcal{O}'\|$, we imagine that for each job J that has scheduled by \mathcal{A}' , we can discharge some credits at a certain rate whenever \mathcal{A}' schedules J . Precisely, we define the rate $\sigma(J)$ to be $\frac{4s}{s-1} \rho(J)$. I.e., we can discharge a total of $\sigma(J)T(J)$ credits due to J .

The rest of this section is divided into two parts. The first part, comprising Lemmas 2.17 and 2.18 and Corollary 2.19,

shows that $\|\mathcal{O}'\|$ is no more than the total amount of credits discharged due to the jobs in $\mathcal{J}(\mathcal{A}')$, which is exactly $\sum_{J \in \mathcal{J}(\mathcal{A}')} \sigma(J)T(J)$. The second part shows that $\|\mathcal{A}'\| \geq \sum_{J \in \mathcal{J}(\mathcal{A}')} \sigma(J)T(J)$. Combining both parts, we can conclude that $\|\mathcal{A}'\| \geq \|\mathcal{O}'\|$.

Lemma 2.17. *Consider any job J_0 scheduled by \mathcal{O}' . For any job $J \in \mathcal{J}(\mathcal{A}')$ such that $T_{J_0}(J) \geq 0$, $\rho(J) \geq \rho(J_0)/2$.*

Proof. Let J_0 be any job scheduled by \mathcal{O}' . By definition, J_0 cannot be completed by \mathcal{A} . Consider any job $J \in \mathcal{J}(\mathcal{A}')$ such that $T_{J_0}(J) \geq 0$. Let t be any time when J_0 is fresh and \mathcal{A}' schedules J as the primary job. By Lemma 2.15, at time t , \mathcal{A} is scheduling a job J' with density at least $\rho(J_0)/2$. Note that J may not be equal to J' . Nevertheless, we can argue that J' must be scheduled by \mathcal{A}' at time t .

- If $J' = J_0$, then J' is not completed by \mathcal{A} . By the definition of \mathcal{A}' , J' is left in \mathcal{A}' .
- Suppose $J' \neq J_0$. By Lemma 2.14, J' is also not equal to any other job scheduled by \mathcal{O} . Again, by the definition of \mathcal{A}' , J' is left in \mathcal{A}' .

Since J is the primary job scheduled by \mathcal{A}' at time t , we have $\rho(J) \geq \rho(J') \geq \rho(J_0)/2$. \square

Lemma 2.18. *For any job J_0 in \mathcal{O}' ,*

$$\sum_{J \in \mathcal{J}(\mathcal{A}')} \sigma(J)T_{J_0}(J) \geq v(J_0) .$$

Proof. Note that J_0 , being a job scheduled by \mathcal{O}' , cannot be completed by \mathcal{A} . By Lemma 2.15, \mathcal{A} must schedule at least one job at any time while J_0 is fresh. By Lemma 2.14, every such job is not scheduled by \mathcal{O} and must be found in \mathcal{A}' . Therefore, $\sum_{J \in \mathcal{J}(\mathcal{A}')} T_{J_0}(J)$ is equal to the length of the fresh period of J_0 . By definition, J_0 is fresh for a period of length $(1-r)p(J_0) = \frac{1}{2}(1-\frac{1}{s})p(J_0)$. Thus, $\sum_{J \in \mathcal{J}(\mathcal{A}')} T_{J_0}(J) = \frac{1}{2}(1-\frac{1}{s})p(J_0)$. Furthermore,

$$\begin{aligned} & \sum_{J \in \mathcal{J}(\mathcal{A}')} \sigma(J)T_{J_0}(J) \\ &= \sum_{J \in \mathcal{J}(\mathcal{A}')} \frac{4s}{s-1} \rho(J)T_{J_0}(J) \\ &\geq \frac{4s}{s-1} \sum_{J \in \mathcal{J}(\mathcal{A}')} \frac{\rho(J_0)}{2} T_{J_0}(J) \quad (\text{by Lemma 2.17}) \\ &= \frac{2s}{s-1} \rho(J_0) \sum_{J \in \mathcal{J}(\mathcal{A}')} T_{J_0}(J) \\ &= \frac{2s}{s-1} \rho(J_0) \cdot \frac{1}{2} \left(1 - \frac{1}{s}\right) p(J_0) \end{aligned}$$

$$\begin{aligned}
 &= \rho(J_0)p(J_0) \\
 &= v(J_0) .
 \end{aligned}$$

□

Corollary 2.19. $\sum_{J \in \mathcal{J}(\mathcal{A}')} \sigma(J)T(J) \geq \|\mathcal{O}'\|.$

Proof. At any time t , Lemma 2.14 implies that among all jobs scheduled by \mathcal{O} , there is at most one which is fresh at t . Thus, for each job $J \in \mathcal{J}(\mathcal{A}')$, $T(J) \geq \sum_{J_0 \in \mathcal{O}'} T_{J_0}(J)$. We have

$$\begin{aligned}
 \sum_{J \in \mathcal{J}(\mathcal{A}')} \sigma(J)T(J) &\geq \sum_{J \in \mathcal{J}(\mathcal{A}')} \sigma(J) \sum_{J_0 \in \mathcal{O}'} T_{J_0}(J) \\
 &\geq \sum_{J_0 \in \mathcal{O}'} \sum_{J \in \mathcal{J}(\mathcal{A}')} \sigma(J)T_{J_0}(J) .
 \end{aligned}$$

By Lemma 2.18, we conclude that

$$\sum_{J_0 \in \mathcal{O}'} \sum_{J \in \mathcal{J}(\mathcal{A}')} \sigma(J)T_{J_0}(J) \geq \sum_{J_0 \in \mathcal{O}'} v(J_0) = \|\mathcal{O}'\| . \quad \square$$

To prove $\|\mathcal{A}'\| \geq \sum_{J \in \mathcal{J}(\mathcal{A}')} \sigma(J)T(J)$, we consider the following amortization scheme on \mathcal{A}' . We associate an account with each job in $\mathcal{J}(\mathcal{A}')$, all having zero initial balance. Credits are put into or removed from these accounts in accordance to the way \mathcal{A}' schedules the jobs. See Figure 5 for details.

- When a job J is completed, $v(J)$ is deposited into the account of J .
- Whenever \mathcal{A}' schedules J , we withdraw credits from the account of J in two ways:
 - Credits are discharged at rate $\sigma(J)$.
 - Credits are transferred to the account of each idling job J' in the pool P at the rate $I(J')$, where $I(J') = \frac{2s}{s-1} \rho(J')$.

Denote $\Psi(J)$ as the final balance of the account of each job J in $\mathcal{J}(\mathcal{A}')$. Note that the sum of all deposits is exactly $\|\mathcal{A}'\|$, and the sum of all discharges is $\sum_{J \in \mathcal{J}(\mathcal{A}')} \sigma(J)T(J)$. That is,

$$\|\mathcal{A}'\| = \sum_{J \in \mathcal{J}(\mathcal{A}')} \Psi(J) + \sum_{J \in \mathcal{J}(\mathcal{A}')} \sigma(J)T(J) .$$

Corollary 2.19 states that $\sum_{J \in \mathcal{J}(\mathcal{A}')} \sigma(J)T(J) \geq \|\mathcal{O}'\|$. In the rest of this section, we will show that if $s \geq 7$, every account has a non-negative final balance and $\sum_{J \in \mathcal{J}(\mathcal{A}')} \Psi(J) \geq 0$. Then we can conclude that $\|\mathcal{A}'\| \geq \|\mathcal{O}'\|$ (see Theorem 2.22).

The following observation is crucial to the proof of the fact that all accounts have non-negative balance. By definition, each job completed by \mathcal{A}' will receive credits equal to the value of the job, which are enough to pay off for all discharges and transfers (see Lemma 2.21). The nontrivial part is concerned with those jobs $J \in \mathcal{J}(\mathcal{A}')$ that miss deadline. Note that J must idle for a long time when it is in the pool P . Note that whenever J idles after being added into P , \mathcal{A} schedules two other jobs. By Lemma 2.14, \mathcal{O} cannot schedule both of these two jobs. In other words, at least one of these two jobs, say, J_a , is not completed by \mathcal{O} . By the definition of \mathcal{A}' , J_a is left in \mathcal{A}' . When \mathcal{A}' schedules J_a and J is idle, credits are transferred from the account of J_a to the account of J . J thus receives credits transferred from other jobs that are scheduled while J is idle in P . More precisely, the following lemma shows that J receives at least $v(J)$ credits.

Lemma 2.20. *For each job J that is scheduled by \mathcal{A}' , at least $v(J)$ credits have been put into the account of J .*

Proof. If J is completed by \mathcal{A}' , the lemma holds by the definition. Now suppose J entered P but is eventually discarded. Then J has been idle in P for a period of at least $\frac{1}{2}(1 - \frac{1}{s})p(J)$. At these times it receives credits from at least one running job at the rate $I(J)$, so it eventually receives at least $\frac{1}{2}(1 - \frac{1}{s})p(J)I(J) = v(J)$ credits. □

We then bound the amount of credits that is removed, (i.e. discharges or transferred) from the account of each job J . When $s \geq 7$, we show that the remaining balance is non-negative.

Lemma 2.21. *Assume that $s \geq 7$. Then for any job J , $\Psi(J) \geq 0$.*

Proof. We only consider jobs that have been scheduled since credits are only removed from the account of a job when the job is being scheduled. We first give an upper bound to the rate of transfer from J to idling jobs. The algorithm ensures that whenever J is scheduled, it is denser than all idling jobs in P . By Lemma 2.16, each job in P is at least twice as dense as the next dense job in P , except that the second least dense job need only be at least as dense as the least dense job. Therefore, if there are i idling jobs, the rate of transfer is at most

$$\begin{aligned}
 &\frac{2s}{s-1} \frac{\rho(J)}{2} + \frac{2s}{s-1} \frac{\rho(J)}{2^2} + \dots \\
 &\quad + \frac{2s}{s-1} \frac{\rho(J)}{2^{i-1}} + \frac{2s}{s-1} \frac{\rho(J)}{2^{i-1}} \\
 &\leq \frac{2s}{s-1} \rho(J) .
 \end{aligned}$$

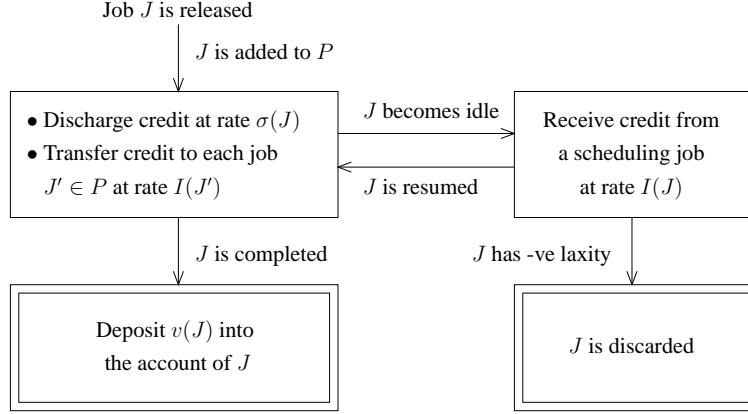


Figure 5: The transfer of credits during the life-span of a job

The total rate of transfer and discharges is thus at most $\sigma(J) + \frac{2s}{s-1} \rho(J)$. Now consider the final balance $\Psi(J)$.

$$\begin{aligned}
 \Psi(J) &\geq v(J) - T(J) \left[\sigma(J) + \frac{2s}{s-1} \rho(J) \right] \\
 &\geq p(J)\rho(J) - \frac{p(J)}{s} \left[\frac{4s}{s-1} \rho(J) + \frac{2s}{s-1} \rho(J) \right] \\
 &= p(J)\rho(J) \cdot \frac{s-7}{s-1} \\
 &\geq 0 \quad (\text{assume } s \geq 7) \quad \square
 \end{aligned}$$

We thus have the following main theorem.

Theorem 2.22. *Assume that the algorithm is using two speed- s processors, where $s \geq 7$. Then $\|\mathcal{A}'\| \geq \|\mathcal{O}'\|$.*

Proof.
$$\|\mathcal{A}'\| = \sum_{J \in \mathcal{J}(\mathcal{A}')} \Psi(J) + \sum_{J \in \mathcal{J}(\mathcal{A}')} \sigma(J)T(J) \geq \sum_{J \in \mathcal{J}(\mathcal{A}')} \Psi(J) + \|\mathcal{O}'\| \geq \|\mathcal{O}'\|. \quad \square$$

Corollary 2.23. *There exists a speed-14 optimal algorithm for UFS- k when all jobs are tight.*

2.4 Concluding Remarks

In this chapter, we have showed two lower bound results on the speed requirement for optimality: For UFS-1, no speed- s optimal algorithm exist unless $s \geq \phi$; for UFS- k , no speed- s optimal algorithm exist unless $s \geq 2$. We have also presented a speed- $O(1)$ optimal algorithm for UFS- k when all jobs are tight. Prior to our work, only a speed- $O(\log k)$ optimal algorithm is known.

3 Scheduling with Additional Processors

This chapter studies on-line algorithms using additional processors for the problems UFS-1 and UFS- k . In other words, we compare on-line algorithms that use multiple speed-1 processors with an off-line algorithm using one speed-1 processor.

Without using additional resources, the best competitive ratio has a matching upper and lower bound: 4 for UFS-1 [3, 4] and $(1 + \sqrt{k})^2$ for UFS- k [3, 22]. For UFS-1, if additional processors are allowed, Baruah [2] gave a m -processor $m/(m-1)$ -competitive algorithm. For example, if $m = 2$, the competitive ratio is improved from 4 to 2. No 1-competitive or optimal algorithm based on additional processors has been heard, let alone UFS- k . In this chapter, we present the first result on attaining constant competitive ratio and optimality using only additional processors, and we show that these results are tight up to a constant factor.

Our new algorithms are based on the Earliest Deadline First algorithm (EDF). A scheduler using EDF always runs the job with the earliest deadline. In practice, EDF is often supplemented with some kind of admission control to avoid excessive preemption when the system is overloaded. We denote EDF-AC as EDF with the following form of admission control: Upon release, a job is tested in order to get admitted for EDF scheduling. The test simply checks whether the new job together with the previously admitted jobs can all be completed by their deadlines using an EDF schedule.

The rest of this chapter is organized as follows. First, we give a two-processor optimal algorithm EDF-Plus for UFS-1 in section 3.1. Then we show in section 3.2 a simple extension of EDF-Plus can attain a $2 \lceil \log k \rceil$ -processor 2-competitive algorithm for UFS- k . This result is asymptotically tight, as we show that no algorithm is w -processor c -competitive for any constant c unless $w = \Omega(\log k)$. In section 3.3, we present a 4-processor optimal algorithm EDF-MSp for UFS-1. EDF-MSp can also be extended to give a $4 \lceil \log k \rceil$ -processor optimal algorithm for UFS- k .

3.1 The EDF-Plus algorithm

In this section we discuss a new algorithm called EDF-Plus which is two-processor optimal for UFS-1. It is known that EDF (and EDF-AC) is one-processor optimal for underloaded systems [9]. Yet this is not true for overloaded systems. Intuitively, it is too difficult for an on-line algorithm to select the right jobs so as to maximize the overall processing time. For example, EDF-AC would make a mistake in rejecting a long job due to the earlier admission of a shorter job with

- (1) Initialization: $AC_Q \leftarrow \emptyset$
- (2)
- (3) When job J is released:
- (4) let J_{M_p} denote the job running in M_p ;
- (5) **if** M_e can complete all jobs in $AC_Q \cup \{J\}$ using EDF
- (6) $AC_Q \leftarrow AC_Q \cup \{J\}$;
- (7) M_e runs the job with the earliest deadline job in AC_Q
- (8) **else if** M_p is idle **or** $p(J) > p(J_{M_p})$
- (9) J_{M_p} is discarded and M_p runs J
- (10) **else**
- (11) J is discarded
- (12)
- (13) When M_e completes job J :
- (14) $AC_Q \leftarrow AC_Q - \{J\}$
- (15) let J_{M_p} denote the job running in M_p ;
- (16) **if** M_e can complete all jobs in $AC_Q \cup \{J_{M_p}\}$ using EDF
- (17) $AC_Q \leftarrow AC_Q \cup \{J_p\}$ // M_p becomes idle
- (18) M_e runs the job with the earliest deadline in AC_Q

Algorithm 3.1: The EDF-Plus algorithm

close deadline. We improve EDF-AC based on a simple idea. When EDF-AC mistakenly rejects a job, we give the job a second chance by scheduling it in another processor temporarily; after a while, the remaining processing time will get smaller and hopefully, the job can get admitted by EDF-AC again. Thus, the enhanced EDF-AC will be more productive.

The above observation leads us to use two processors, denoted M_e and M_p , in the algorithm EDF-Plus. M_e schedules jobs using EDF-AC. Once M_e admits a job, the job is guaranteed to be completed. A rejected job is considered by M_p immediately. M_p aims at scheduling a rejected job temporarily. The job in M_p will repeatedly attempt to migrate to M_e , by going through the admission control of M_e whenever M_e completes a job. Note that at any time, there may be more than one job rejected by M_e ; yet M_p only works on the job with the longest processing time as soon as it is rejected by M_e . All other rejected jobs from M_e are given up immediately. Algorithm 3.1 gives the details of EDF-Plus.

Theorem 3.1. EDF-Plus is two-processor optimal for UFS-1.

In the remainder of this section, we prove Theorem 3.1 by contradiction. Assume that EDF-Plus is not optimal for some job sequence. Let \mathcal{I} be the one containing the fewest jobs. Without loss of generality, suppose the first job is released at time 0. We first establish that for such \mathcal{I} , EDF-Plus keeps M_e busy over one continuous period (see Lemma 3.3). Then we

show in Lemma 3.4 an interesting property of the job J_ℓ in \mathcal{I} that has the latest deadline. Using these lemmas, we show that the total processing time of jobs completed by M_e is more than $d(J_\ell)$ (see Lemma 3.5). Note that jobs of \mathcal{I} can only be scheduled within the period $[0, d(J_\ell)]$. Thus, an off-line algorithm, using one processor, obtains a total value (processing time) of at most $d(J_\ell)$. This contradicts that EDF-Plus is not optimal for \mathcal{I} and we complete the proof of Theorem 3.1.

Fact 3.2. At any time, if M_e is idle, then AC-Q (the queue storing all admitted jobs to be completed by M_e) is empty and M_p is idle.

Lemma 3.3. *In the course of scheduling \mathcal{I} , M_e is busy over exactly one continuous period.*

Proof. Assume that M_e is busy over two or more disjoint periods. Let t_1 be the start time of the last busy period. Partition \mathcal{I} into two parts, one for jobs with release time before t_1 and one for the rest. Since EDF-Plus is not optimal for input \mathcal{I} , at least one of the two parts gives a job sequence that EDF-Plus is not optimal. This contradicts that \mathcal{I} contains the fewest jobs. \square

We need the following notion to analyze J_ℓ , the job with the latest deadline.

Definition. Consider any time t when M_e rejects a job J (see line 5 and 16 in Alg. 3.1). That is, if M_e uses EDF to schedule J together with the jobs admitted before t , some jobs J_o miss their deadlines. Any such jobs J_o is said to *repudiates* J at t . Note that a job can only repudiate itself or jobs with earlier deadlines.

Lemma 3.4. *In the course of scheduling \mathcal{I} , there is at least one time when J_ℓ repudiates a job.*

Proof. Suppose on the contrary that J_ℓ never repudiates any job. J_ℓ and all other jobs do not repudiate J_ℓ when it is released; thus, J_ℓ must be admitted for M_e . Consider any moment after J_ℓ is admitted. Any newly released job, if rejected by M_e , must be repudiated by a job other than J_ℓ . Recall that M_e is running EDF-AC and J_ℓ has the latest deadline. If we remove J_ℓ from \mathcal{I} , M_e will not admit more jobs and EDF-Plus loses the processing time of J_ℓ without gaining anything. On the other hand, the optimal off-line algorithm loses at most the processing time of J_ℓ . Thus, $\mathcal{I} - \{J_\ell\}$ is a job sequence for which EDF-Plus is not optimal. This contradicts that \mathcal{I} contains the fewest jobs. \square

Lemma 3.5. *The value obtained by EDF-Plus in scheduling \mathcal{I} is more than $d(J_\ell)$.*

Proof. By Lemma 3.4, J_ℓ repudiates some job J at some time t . Note that $r(J_\ell) \leq t \leq d(J_\ell)$ and M_e must be busy at time t . Furthermore, by Lemma 3.3, M_e is busy throughout the period $[0, t]$. By definition, at time t , using EDF to schedule the jobs currently found in AC-Q and J will cause J_ℓ to miss its deadline. In other words, M_e is committed to process admitted jobs up to a time later than $d(J_\ell) - p(J)$, attaining a total value of more than $d(J_\ell) - p(J)$.

Next, we show that J or another even longer rejected job will be completed by EDF-Plus. After M_e rejects J at time t , there are three possible scenarios: (1) J is scheduled to completion on M_p ; (2) J is scheduled on M_p and later migrates to M_e ; or (3) J is discarded before its deadline by M_p due to the presence of another rejected job with longer processing time. In the last case, EDF-Plus guarantees that a rejected job with longer processor time will eventually be completed. The value obtained in scheduling rejected jobs is at least $p(J)$.

Therefore, the total value obtained by EDF-Plus for scheduling \mathcal{I} is more than $d(J_\ell) - p(J) + p(J) = d(J_\ell)$. \square

3.2 Constant Competitiveness

In this section we show the additional processor requirement for achieving constant competitiveness is $\Theta(\log k)$. In fact, EDF-Plus is readily to give a performance guarantee for general value density as follows.

Lemma 3.6. *EDF-Plus is two-processor k -competitive for UFS- k .*

Proof. EDF-Plus ignores the value density of the jobs, yet Theorem 3.1 guarantees that its total processing time on completed jobs matches that of any off-line algorithm. The best an off-line algorithm can do is to schedule jobs all with value density k . Result follows. \square

Theorem 3.7. *There exists a $2 \lceil \log k \rceil$ -processor 2-competitive algorithm for UFS- k .*

Proof. Consider the following $2 \lceil \log k \rceil$ -processor algorithm. Partition the jobs into $\lceil \log k \rceil$ groups, where the i -th group contains all the jobs with value density $2^{i-1} \leq \rho(J) \leq 2^i$. Each group is given two processors executing EDF-Plus independently. Within each group, the value densities differ by at most a factor of 2, so the two processors match the value obtained by any off-line algorithm for jobs of this group. Therefore, the $2 \lceil \log k \rceil$ processors together can match the value obtained by any off-line algorithm for jobs with value densities in $[1, k]$. \square

Theorem 3.8. *For UFS- k , any w -processor c -competitive algorithm satisfies the relation that $c \geq \frac{1}{2} \sqrt[w]{k}$.*

Before we prove this theorem, let us consider its consequence. Suppose there is a w -processor c -competitive algorithm for some constant c . By Theorem 3.8, we have $c \geq \frac{1}{2} \sqrt[w]{k}$, or equivalently, $w \geq \log_{2c} k$. Therefore, $w = \Omega(\log k)$, and we obtain the asymptotically tight lower bound.

The rest of this section is devoted to proving Theorem 3.8. Let \mathcal{A} be a w -processor algorithm where $w < \log k$. Without loss of generality, we assume that $k > 2$ and $w \geq 1$. Below we construct a job sequence to make \mathcal{A} perform poorly. Let $d = \frac{1}{2} \sqrt[w]{k}$ and $\varepsilon = \frac{1}{wdk}$. Define \mathcal{J} to be a set of $w + 1$ tight jobs, each belonging to a distinct category defined below. Notice that $2d\varepsilon = \frac{2}{wk} < 1$, and the job value is decreasing from Category 0 to Category w .

category	value density	processing time	value
0	1	1	1
1	$2d$	ε	$2d\varepsilon$
2	$(2d)^2$	ε^2	$(2d\varepsilon)^2$
		\vdots	
w	$(2d)^w$	ε^w	$(2d\varepsilon)^w$

Table 2: The input job set \mathcal{J} for uniprocessor scheduling

We consider an adversary which releases the job sequence in stages and schedules the jobs with one processor. The first stage begins at time 0 when \mathcal{J} is released. Since there are $w + 1$ jobs and \mathcal{A} uses only w processors, there is a job not running by \mathcal{A} at time 0. Denote this job as J_0 . The adversary schedule J_0 using its only processor and release another \mathcal{J} when it completes J_0 .

In general, at the beginning of the i -th stage, the adversary releases another \mathcal{J} and chooses a job J_i to run as follows. Note that at that time, the jobs chosen by \mathcal{A} may not have distinct value densities as \mathcal{A} may continue running some jobs released in previous stages. Let ℓ be the smallest category such that \mathcal{A} runs only ℓ jobs in Category 0 to ℓ . That is, \mathcal{A} runs ℓ jobs in Category 0 to $\ell - 1$, but not any in Category ℓ . Let J_i be the job in Category ℓ just released. Denote $\alpha_i = \ell$ as the category of J_i . The last stage lasts for a time period of 1; this ensures that the deadline of every job released so far is no later than the end of the last stage.

Fact 3.9. In the i -th stage, the adversary completes job J_i , obtaining a value of $v(J_i) = (2d\varepsilon)^{\alpha_i}$.

Since all jobs are tight, if a job is not scheduled to run on a processor upon release, it will definitely miss its deadline. In other words, in the middle of a stage, it makes no sense for a processor to switch to another job. Thus, we can assume that within a stage, a processor runs at most one job.

Lemma 3.10. In the i -th stage (except the last one), the value obtained by \mathcal{A} is at most $\frac{1}{d}(2d\varepsilon)^{\alpha_i}$, i.e., $v(J_i)/d$.

Proof. By the definition of α_i , throughout the i -th stage \mathcal{A} runs α_i jobs in Category 0 to $\alpha_i - 1$ and at most $w - \alpha_i$ jobs in Category $\alpha_i + 1$ to w .

Let us first consider jobs in Category $\alpha_i + 1$ to w . We show that the total value due to such jobs is at most $\frac{1}{d} \varepsilon^{\alpha_i}$. The duration of the i -th stage is ε^{α_i} , which is long enough to complete any job in Category $\alpha_i + 1$ to w . Each processor running a job in Category $\ell \geq \alpha_i + 1$ gives a value of $(2d\varepsilon)^\ell \leq (2d\varepsilon)^{\alpha_i + 1}$. Thus, the total value obtained by such processors is bounded by

$$\begin{aligned} w(2d\varepsilon)^{\alpha_i + 1} &\leq w(2d)^w \varepsilon^{\alpha_i + 1} \\ &= w(\sqrt[w]{k})^w \frac{1}{wdk} \varepsilon^{\alpha_i} \\ &= \frac{1}{d} \varepsilon^{\alpha_i}. \end{aligned}$$

Next, we consider the jobs in Category 0 to $\alpha_i - 1$. Recall that these α_i jobs may not have distinct value densities. Nevertheless, by the definition of α_i , the sum of their value densities is at most $1 + 2d + (2d)^2 + \dots + (2d)^{\alpha_i - 1}$. Assuming all these jobs are running throughout the i -th stage, the value obtained is at most

$$\begin{aligned} &[1 + 2d + (2d)^2 + \dots + (2d)^{\alpha_i - 1}] \varepsilon^{\alpha_i} \\ &= \frac{(2d)^{\alpha_i} - 1}{2d - 1} \varepsilon^{\alpha_i} \\ &\leq \frac{(2d)^{\alpha_i} - 1}{d} \varepsilon^{\alpha_i}. \end{aligned}$$

Summing the above two parts together, we conclude that the value obtained by \mathcal{A} during the i -th stage is at most $\frac{1}{d} \varepsilon^{\alpha_i} + \frac{(2d)^{\alpha_i} - 1}{d} \varepsilon^{\alpha_i} = \frac{1}{d} (2d\varepsilon)^{\alpha_i}$, which completes the proof. \square

In the last stage, the best \mathcal{A} can do is to complete the first w jobs just released, obtaining a total value of at most w . The adversary on the other hand completes the job with value 1.

Consider an instance of the above job sequence that consists of $h + 1$ stages. Let \mathcal{O} be the value obtained by the adversary during the first h stages. By Lemma 3.10, the competitive ratio of \mathcal{A} is at least $(\mathcal{O} + 1)/(\mathcal{O}/d + w)$. Notice that w is a constant, and we can choose a sufficiently large h to make \mathcal{O} arbitrarily large and the ratio arbitrarily close to d . Therefore, the competitive ratio of \mathcal{A} has a lower bound of d , which is defined as $\frac{1}{2} \sqrt[w]{k}$. This completes the proof of Theorem 3.8.

3.3 Optimality

In this section we first present an algorithm called EDF-MSp which is 4-processor optimal for UFS-2. Then we show that for UFS- k , a simple extension of EDF-MSp can give a $4 \lceil \log k \rceil$ -processor optimal algorithm. EDF-MSp uses four processors, divided into two *bands*, each containing two processors. When a job is released, it is first considered by Band 1, which is running EDF-Plus. If Band 1 discards the job (at line 9 or line 11 in Alg. 3.1), the job is passed to Band 2.

For any job sequence \mathcal{I} , let \mathcal{A}_1 and \mathcal{O} be the sets of jobs completed by EDF-Plus and an optimal off-line algorithm OPT. For any job set \mathcal{S} , denote $\|\mathcal{S}\|$ as the sum of the value of all jobs in \mathcal{S} . We overload the symbol $p(\mathcal{S})$ to denote the sum of the processing time of all jobs in \mathcal{S} . Recall that EDF-Plus guarantees that $p(\mathcal{A}_1) \geq p(\mathcal{O})$. Though jobs in \mathcal{O} may have higher value, the importance ratio is at most two and $\|\mathcal{O}\|$, the total value of \mathcal{O} , is at most $2p(\mathcal{O})$. Optimality can be achieved if the Band 2 processors can complete a subset \mathcal{A}_2 of jobs discarded by EDF-Plus with sufficient processing time, say, $p(\mathcal{A}_2) \geq p(\mathcal{O})$. Then we can conclude that $\|\mathcal{A}_1\| + \|\mathcal{A}_2\| \geq \|\mathcal{O}\|$. Yet providing such a guarantee on $p(\mathcal{A}_2)$ seems to be very difficult. In fact, our algorithm takes advantage of a less demanding requirement, namely, $p(\mathcal{A}_2) \geq p(\mathcal{O} - \mathcal{A}_1 - \mathcal{A}_2)$.

Lemma 3.11. *Let $\mathcal{O}' = \mathcal{O} - \mathcal{A}_1 - \mathcal{A}_2$. Suppose $p(\mathcal{A}_2) \geq p(\mathcal{O}')$. Then $\|\mathcal{A}_1\| + \|\mathcal{A}_2\| \geq \|\mathcal{O}\|$.*

Proof. Denote $\mathcal{S}_1 = \mathcal{A}_1 \cap \mathcal{O}$ and $\mathcal{S}_2 = \mathcal{A}_2 \cap \mathcal{O}$. Note that $\mathcal{O}' = \mathcal{O} - \mathcal{S}_1 - \mathcal{S}_2$ and $\|\mathcal{O}\| = \|\mathcal{S}_1\| + \|\mathcal{S}_2\| + \|\mathcal{O}'\|$. Furthermore, let $\mathcal{A}'_1 = \mathcal{A}_1 - \mathcal{S}_1$, and let $\mathcal{A}'_2 = \mathcal{A}_2 - \mathcal{S}_2$. Since $p(\mathcal{A}_1) \geq p(\mathcal{O})$, we have $p(\mathcal{A}'_1) = p(\mathcal{A}_1) - p(\mathcal{S}_1) \geq p(\mathcal{O}) - p(\mathcal{S}_1) = p(\mathcal{O}') + p(\mathcal{S}_2)$. Moreover, $p(\mathcal{A}_2) \geq p(\mathcal{O}')$ and thus $p(\mathcal{A}'_2) = p(\mathcal{A}_2) - p(\mathcal{S}_2) \geq p(\mathcal{O}') - p(\mathcal{S}_2)$. In summary, we have $p(\mathcal{A}'_1) + p(\mathcal{A}'_2) \geq 2p(\mathcal{O}') \geq \|\mathcal{O}'\|$. Therefore, $\|\mathcal{A}_1\| + \|\mathcal{A}_2\| \geq \|\mathcal{S}_1\| + \|\mathcal{S}_2\| + p(\mathcal{A}'_1) + p(\mathcal{A}'_2) \geq \|\mathcal{O}\|$. \square

Below we derive an algorithm called MSp for Band 2 so that $p(\mathcal{A}_2) \geq p(\mathcal{O}')$, where $\mathcal{O}' = \mathcal{O} - \mathcal{A}_1 - \mathcal{A}_2$. First of all, we note that job J passed to MSp is discarded by EDF-Plus either at $r(J)$ or strictly after $r(J)$. For the latter case, we observe the following property.

Fact 3.12. *If a job J is discarded by EDF-Plus at time $t > r(J)$, it has been running on M_p during $[r(J), t]$.*

We denote the two processors of MSp as M_r and M_d . Details of MSp are shown in Algorithm 3.2. Intuitively, for every job J discarded by Band 1, we hope that either J is completed in Band 2, or M_r is busy during the entire period $[r(J), d(J)]$. M_r attempts to schedule and complete any job discarded by

```

(1) Initialization: SLACK_Q  $\leftarrow$   $\emptyset$ 
(2) // A job in SLACK_Q waits until its slack time is zero
(3)
(4) When job  $J$  is passed to band 2:
(5) let  $J_r$  is the job running in  $M_r$ ;
(6) if  $M_r$  is idle or  $r(J) < r(J_r)$ 
(7)   SLACK_Q  $\leftarrow$  SLACK_Q  $\cup$   $\{J_r\}$ 
(8)    $M_r$  runs  $J$ 
(9) else
(10)  SLACK_Q  $\leftarrow$  SLACK_Q  $\cup$   $\{J\}$ 
(11)
(12) When  $M_r$  completes job  $J$ :
(13) if SLACK_Q  $\neq$   $\emptyset$ 
(14)  SLACK_Q  $\leftarrow$  SLACK_Q  $-$   $\{J'\}$  where  $J'$  is chosen
(15)  arbitrarily from SLACK_Q
(16)   $M_r$  runs  $J'$ 
(17) else if  $M_d$  is working on a job  $J_d$ 
(18)   $M_r$  runs  $J_d$  //  $M_d$  becomes idle
(19)
(20) When job  $J \in$  SLACK_Q has zero slack time:
(21) let  $J_d$  is the job running in  $M_d$ ;
(22) SLACK_Q  $\leftarrow$  SLACK_Q  $-$   $\{J\}$ 
(23) if  $M_d$  is idle or  $d(J) > d(J_d)$ 
(24)   $M_d$  runs  $J$ ;  $J_d$  is discarded
(25) else
(26)   $J$  is discarded

```

Algorithm 3.2: MSp – the algorithm for Band 2.

Band 1. However, to guarantee productivity as early as possible, a discarded job from Band 1 with an earlier release time can preempt the current job in M_r . On the other hand, for any job J that is not scheduled by M_r , we give J a second chance by scheduling it in M_d temporarily and J may migrate to M_r whenever M_r becomes idle. This ensures M_r to be busy as long as possible. M_d uses the zero slack time strategy, i.e., M_d runs a job J only when J 's slack time is zero; if there are more than one such job, M_d runs the one with the latest deadline. Similar to the two processors of EDF-Plus, M_r and M_d have the following relationship.

Fact 3.13. *At any time, if M_r is idle, then SLACK_Q is empty and M_d is idle.*

The crux of the analysis of Band 2 is captured by the following theorem. Recall that with respect to a given job sequence \mathcal{I} , \mathcal{A}_1 and \mathcal{A}_2 denote the set of jobs completed by Band 1 and Band 2, respectively, and \mathcal{I}' denotes the set of jobs passed to but not completed by Band 2 (i.e., $\mathcal{I}' = \mathcal{I} - \mathcal{A}_1 - \mathcal{A}_2$). Furthermore, we need the following definition.

Definition. *The span of a set \mathcal{S} of jobs is the union of the*

spans of all the jobs in \mathcal{S} . (E.g., the union of the spans [3, 6] and [5, 8] is [3, 8].) Furthermore, let $sp(\mathcal{S})$ be the total time included in the span of \mathcal{S} .

Theorem 3.14. $p(\mathcal{A}_2) \geq sp(\mathcal{I}')$.

Before proving Theorem 3.14, we note that Theorem 3.14 guarantees that EDF-MSp is a four-processor optimal algorithm for scheduling jobs with value densities in the range [1, 2].

Corollary 3.15. (i) $p(\mathcal{A}_2) \geq sp(\mathcal{O}')$; (ii) $\|\mathcal{A}_1\| + \|\mathcal{A}_2\| \geq \|\mathcal{O}\|$.

Proof. As $\mathcal{O}' \subseteq \mathcal{I}'$ and OPT schedules at most one job at a time, we have $p(\mathcal{O}') \leq sp(\mathcal{I}')$. By Theorem 3.14, $p(\mathcal{A}_2) \geq sp(\mathcal{I}') \geq p(\mathcal{O}')$. Then, by Lemma 3.11, we conclude that $\|\mathcal{A}_1\| + \|\mathcal{A}_2\| \geq \|\mathcal{O}\|$. \square

We prove Theorem 3.14 via the following three lemmas.

Lemma 3.16. *Let J be a job passed to Band 2 at time $t > r(J)$. Then any job J' with $r(J') \leq r(J)$, if passed to Band 2, must be passed on or before $r(J)$.*

Proof. Assume on the contrary that there is a job J' passed to Band 2 at time t' such that $r(J') \leq r(J) < t'$. Consider the nonempty interval $[r(J), \min(t, t')]$. By applying Fact 3.12 to J and J' , we obtain the contradiction that M_p has been running J and J' during this interval. \square

Definition. At any time t , EDF-MSp is said to be *productive* on \mathcal{A}_2 if a job $J \in \mathcal{A}_2$ is running on one of the four processors of EDF-MSp.

Lemma 3.17. *At any time, if M_r is busy, then EDF-MSp is productive on \mathcal{A}_2 .*

Proof. Consider any time t_1 when M_r runs a job J . The lemma holds if J is completed on M_r . It remains to consider the case that J is preempted by another job J' passed to Band 2 at time $t_2 \geq t_1$. By definition of EDF-MSp, $r(J') < r(J)$. We want to show that J' is in \mathcal{A}_2 . Note that $r(J') < r(J) \leq t_1 \leq t_2$. By Fact 3.12, J' is running on M_p during the period $[r(J'), t_2]$. By Lemma 3.16, all jobs passed to Band 2 after $t_2 \geq r(J')$ are released later than J' . Therefore, J' cannot be preempted by these jobs and can run up to completion on M_r . Thus, J' is in \mathcal{A}_2 and EDF-MSp is productive on \mathcal{A}_2 during $[r(J'), t_2]$ and in particular at time t_1 . \square

Lemma 3.18. *Let J be any job discarded by EDF-MSp. Then EDF-MSp is productive on \mathcal{A}_2 throughout the span of J , i.e., $[r(J), d(J)]$.*

Proof. By Lemma 3.17, it suffices to show that M_r is busy during the span of J . We divide the span into three periods (which may not all exist) and argue M_r is busy in each period.

- Consider the period from $r(J)$ to the time t_o when J is passed to Band 2. Suppose $t_o > r(J)$. Assume, for the sake of contradiction, that M_r is idle at a certain time $t \in [r(J), t_o]$. Then all jobs passed to Band 2 before t should have been completed or discarded by t ; otherwise M_r should schedule one at t . In other words, any job J' found in Band 2 after time t must be passed to Band 2 at time after $t > r(J)$. By Lemma 3.16, if $J \neq J'$, then $r(J') > r(J)$. When J is passed to Band 2 at t_o , it can preempt the job currently in M_r (if exist) and will not be preempted afterward. Therefore, J can run up to completion on M_r , contradicting that J is discarded by EDF-MSp.
- As J is discarded eventually, it must have put into SLACK_Q at least once. Consider the period from t_o to the last time t_ℓ when J is removed from SLACK_Q for consideration of M_d . At any time within this period, J is in SLACK_Q or is processed by M_d or M_r . In the first two cases, M_r cannot be idle because of Fact 3.13 (i.e., J is eligible for scheduling on M_r).
- At time t_ℓ , M_d attempts to schedule J . Recall that J is discarded by EDF-MSp. J must be preempted before its deadline. By definition of M_d , this must be due to a job J' with a later deadline. Note that J' may possibly be further preempted or migrated to M_r . In all cases, at any time within the period $[t_\ell, d(J)]$, there is at least one job with deadline on or after $d(J)$ scheduled by either M_r or M_d . In the latter case, by Fact 3.13, M_r must be busy with some other job. \square

We are now ready to prove Theorem 3.14, i.e., $p(\mathcal{A}_2) \geq sp(\mathcal{I}')$.

Proof of Theorem 3.14. First of all, $p(\mathcal{A}_2)$ is at least the total time during which EDF-MSp is productive on \mathcal{A}_2 . Lemma 3.18 ensures that for any job $J \in \mathcal{I}'$, EDF-MSp is productive on \mathcal{A}_2 during the span of J . In other words, EDF-MSp is productive on \mathcal{A}_2 during the span of \mathcal{I}' . Therefore, $p(\mathcal{A}_2) \geq sp(\mathcal{I}')$. \square

EDF-MSp can serve as a building block for handling jobs with general importance ratio.

Theorem 3.19. *There exists a $4 \lceil \log k \rceil$ -processor optimal algorithm for UFS- k .*

Proof. Consider the following $4 \lceil \log k \rceil$ -processor algorithm. Partition the jobs into $\lceil \log k \rceil$ groups, where the i -th group contains all the jobs with value density in the range $[2^{i-1}, 2^i]$.

Each group is given 4 processors executing EDF-MSp independently. Within each group, the value densities differ by at most a factor of 2, so the four processors match the value obtained by any off-line algorithm for jobs of this group. Therefore, the $4 \lceil \log k \rceil$ processors together can match the value obtained by any off-line algorithm for jobs with value densities in $[1, k]$. \square

3.4 Concluding Remarks

In this chapter, we have presented a 2-processor optimal algorithm for UFS-1. Extending this algorithm gave a $2 \lceil \log k \rceil$ -processor 2-competitive algorithm and a $4 \lceil \log k \rceil$ -processor optimal algorithm for UFS- k . These results are asymptotically tight as we have showed no p -processor $O(1)$ -competitive or optimal algorithm exist unless $p = \Omega(\log k)$.

4 Multiprocessor Scheduling

In this chapter, we consider the competitiveness of on-line algorithms using additional resources for MFS- k . I.e., for any integer $m \geq 2$, we are interested in comparing the performance of on-line algorithms using m or more possibly faster processors against an off-line algorithm using m processors.

Without additional resources, Koren and Shasha [21] gave an algorithm called MOCA that has a competitive ratio of $1 + m(k^{1/\psi} - 1)$, where $\psi = \frac{m}{2} \frac{\log k}{\log k + 1}$. They also showed a lower bound of $\frac{k}{k-1} m(k^{1/m} - 1)$. When m tends to infinity, both bounds tend to $O(\log k)$. When slightly faster processors are used, Kalyanasundaram and Pruhs [17] demonstrated a speed- $O(1)$ $O(1)$ -competitive algorithm called SLACKER for UFS- k , and Lam and To [25] extended the algorithm to give a speed- $O(1)$ $O(1)$ -competitive algorithm called MSLACKER for MFS- k . For example, the algorithm is speed-2 21-competitive. However, no result using additional processors is known.

In section 4.1, we extend MOCA to using additional processors and prove that it is $O(\log k)$ -processor $O(1)$ -competitive. We also show that $\Omega(\log k)$ times additional processors (i.e., $\Omega(m \log k)$ processors) are required to achieve constant competitiveness.

The algorithms SLACKER [17] and MSLACKER [25] allow a trade-off between the speed and the competitive ratio. For example, using a speed-3 instead of a speed-2 processor, the competitive ratio can be improved from 21 to 7. In section 4.2, we consider an extension of MSLACKER. We show that to reduce the competitive ratio of MSLACKER, we can use additional speed- s processors instead of increasing the speed requirement s . For instance, when $s = 2$, MSLACKER can be improved from 21-competitive to 5-processor speed-2 5-competitive.

4.1 Competitiveness with Additional Processors

In this section we prove that $\Theta(\log k)$ additional processors are required to achieve constant competitiveness.

4.1.1 Lower bound

Theorem 4.1. *For MFS- k , any w -processor c -competitive algorithm satisfies the relation that $c \geq m(\sqrt[m]{k} - 1)$.*

Let us look at the consequence of the above theorem. Suppose there is a w -processor c -competitive algorithm for some constant c . By Theorem 4.1, we have $c \geq m(\sqrt[m]{k} - 1)$, or equivalently, $w \geq \log k / (m \log(\frac{c+1}{m}))$. Therefore, $w = \Omega(\log k)$, and we obtain the lower bound.

The analysis is very similar to the proof of Theorem 3.8. Given any w -processor algorithm \mathcal{A} , we consider job set \mathcal{J} consisting of $m(mw + 1)$ tight jobs. Denote $\lambda = k^{1/mw}$. The jobs are divided into $mw + 1$ categories, each with m jobs and value densities $1, \lambda, \lambda^2, \dots, \lambda^{mw-1}$, and λ^{mw} . Note that the importance ratio of the jobs is $\lambda^{mw} = k$. We call these jobs Category 0, 1, 2, \dots , $mw - 1$, and mw , respectively. Let ε be a small constant such that $\varepsilon\lambda < 1/k$, i.e., $\varepsilon < k^{-(1+1/mw)}$. The job set \mathcal{J} is shown in Table 3.

category	value density	processing time	value
0	1	1	1
1	λ	ε	$\varepsilon\lambda$
2	λ^2	ε^2	$(\varepsilon\lambda)^2$
		\vdots	
mw	k	ε^{mw}	$k\varepsilon^{mw}$

Table 3: The input job set \mathcal{J} for multiprocessor scheduling

The input is divided into stages. At the beginning of the i -th stage, the adversary releases \mathcal{J} and chooses a category as follows. Let ℓ be the smallest category such that \mathcal{A} runs only ℓ jobs in Category 0 to ℓ . That is, \mathcal{A} runs ℓ jobs in Category 0 to $\ell - 1$, but not any in Category ℓ . Denote $\alpha_i = \ell$. The adversary schedules m Category α_i jobs and at the time of completion it starts the next stage. The last stage lasts for a time period of 1; this ensures that the deadline of every job released is no later than the end of the last stage.

Fact 4.2. *In the i -th stage, the adversary completes m Category α_i jobs, obtaining a value of $m(\varepsilon\lambda)^{\alpha_i}$.*

Since all jobs are tight, if a job is not scheduled to run on a processor upon release, it will definitely miss its deadline. In other words, in the middle of a stage, it makes no sense for a processor to switch to another job. Thus, we can assume that within a stage, a processor runs at most one job.

Lemma 4.3. *In the i -th stage (except the last one), the ratio of the value obtained by the adversary to that by \mathcal{A} is at least $m(\sqrt[m]{k} - 1)$.*

Proof. By the definition of α_i , throughout the i -th stage \mathcal{A} runs α_i jobs in Category 0 to $\alpha_i - 1$ and at most $mw - \alpha_i$ jobs in Category $\alpha_i + 1$ to mw .

Let us first consider the jobs in Category $\alpha_i + 1$ to mw . The duration of the i -th stage is ε^{α_i} , which is long enough to complete any job in Category $\alpha_i + 1$ to mw . Each processor running a job in Category $\ell \geq \alpha_i + 1$ gives a value of $(\varepsilon\lambda)^\ell \leq (\varepsilon\lambda)^{\alpha_i+1}$. Thus, the total value obtained is bounded by

$$(mw - \alpha_i)(\varepsilon\lambda)^{\alpha_i+1}.$$

Next, we consider the jobs in Category 0 to $\alpha_i - 1$. Recall that these α_i jobs may not have distinct value densities. Nevertheless, by the definition of α_i , the sum of their value densities is at most $1 + \lambda + \lambda^2 + \dots + \lambda^{\alpha_i - 1}$. Assuming all these jobs are running throughout the i -th stage, the value obtained by \mathcal{A} is at most

$$[1 + \lambda + \lambda^2 + \dots + \lambda^{\alpha_i - 1}] \varepsilon^{\alpha_i} = \frac{\lambda^{\alpha_i} - 1}{\lambda - 1} \varepsilon^{\alpha_i} .$$

Therefore, the ratio of the value obtained by the adversary and \mathcal{A} is at least

$$\begin{aligned} & \min_{1 \leq \alpha_i \leq mw} \frac{m(\varepsilon\lambda)^{\alpha_i}}{(mw - \alpha_i)(\varepsilon\lambda)^{\alpha_i + 1} + \frac{\lambda^{\alpha_i} - 1}{\lambda - 1} \varepsilon^{\alpha_i}} \\ &= m(\lambda - 1) \cdot \frac{(\varepsilon\lambda)^{\alpha_i}}{\min_{1 \leq \alpha_i \leq mw} (\lambda - 1)(mw - \alpha_i)(\varepsilon\lambda)^{\alpha_i + 1} + (\varepsilon\lambda)^{\alpha_i} - \varepsilon^{\alpha_i}} \\ &= m(\lambda - 1) \cdot \frac{1}{\min_{1 \leq \alpha_i \leq mw} (\lambda - 1)(mw - \alpha_i)\varepsilon\lambda + 1 - \lambda^{-\alpha_i}} . \end{aligned}$$

The denominator is increasing with α_i for $\alpha_i < mw$ since

$$\begin{aligned} & \lambda^{-\alpha_i} - \lambda^{-(\alpha_i + 1)} \\ &= \lambda^{-(\alpha_i + 1)}(\lambda - 1) \\ &\geq \lambda^{-mw}(\lambda - 1) \\ &\geq (\lambda - 1)/k \\ &> (\lambda - 1)\varepsilon\lambda . \end{aligned}$$

Thus, the minimum occurs at $\alpha_i = mw$. The competitive ratio is at least

$$\begin{aligned} & m(\lambda - 1) \frac{1}{1 - \lambda^{-mw}} \\ &> m(\lambda - 1) \\ &= m(\sqrt[mw]{k} - 1) . \quad \square \end{aligned}$$

In the last stage, the best \mathcal{A} can do is to complete the first mw jobs just released, obtaining a total value of at most mw . The adversary on the other hand completes the m jobs with value 1.

Consider an instance of the above job sequence that consists of $h + 1$ stages. Let \mathcal{O} be the value obtained by \mathcal{A} in the first h stages. By Lemma 4.3, the competitive ratio of \mathcal{A} is at least $(m(\sqrt[mw]{k} - 1)\mathcal{O} + m)/(\mathcal{O} + mw)$. Notice that mw is a constant, and we can choose a sufficiently large h to make \mathcal{O} arbitrarily large and the ratio arbitrarily close to $m(\sqrt[mw]{k} - 1)$. Therefore, the competitive ratio of \mathcal{A} has a lower bound of $m(\sqrt[mw]{k} - 1)$. This completes the proof of Theorem 4.1.

```

(1) Initialization:
(2)    $Q \leftarrow \emptyset$  // A job  $J$  in  $Q$  waits for time  $d(J) - p(J)$ 
(3)
(4) When job  $J$  is released:
(5)   if  $J$  can be admitted by a  $Me_i$  for some  $i$ 
(6)      $Me_i$  admits  $J$  and reschedule according to EDF
(7)   else
(8)      $Q \leftarrow Q \cup \{J\}$ 
(9)
(10) When a  $Me_i$  completes all jobs it admitted:
(11)   if  $Md_i$  is running a job  $J$ 
(12)     Switch the roles of  $Me_i$  and  $Md_i$ 
(13)     // I.e.  $Me_i$  becomes  $Md_i$  and vice versa
(14)
(15) When a job  $J \in Q$  arrives time  $d(J) - p(J)$ :
(16)    $Q \leftarrow Q - \{J\}$ 
(17)   if there exists an idling  $Md_i$ 
(18)      $Md_i$  runs  $J$ 
(19)   else
(20)     Denote  $Md_i$  as the  $Md$  running a job  $J_i$  with the
(21)     earliest deadline
(22)     if  $d(J) > d(J_i)$ 
(23)        $Md_i$  runs  $J$  //  $J_i$  is discarded
(24)     // Otherwise  $J$  is discarded
    
```

Algorithm 4.1: Multiprocessor scheduling algorithm (MSA)

4.1.2 Upper bound

We now show that $O(\log k)$ additional processors is sufficient to achieve constant competitiveness. Consider a multiprocessor scheduling algorithm (MSA) with $2wm$ processor (i.e., MSA is a $2w$ -processor algorithm). The processors are divided into wm bands, each consisting of 2 processors denoted as Me and Md . Me schedules jobs using EDF-AC³. A rejected job is considered by Md at time $d(J) - p(J)$, where a later deadline job can preempt an earlier deadline job. The details of MSA is shown in Algorithm 4.1. We prove the competitiveness of MSA in Theorem 4.4, with which we can conclude the upper bound in Corollary 4.5.

Theorem 4.4. *MSA is $2w$ -processor $(1 + \frac{k}{w})$ -competitive for MFS- k .*

Corollary 4.5. *There exists a $2w \lceil \log k \rceil$ -processor $(1 + \frac{2}{w})$ -competitive algorithm for MFS- k .*

Proof. Consider the following $2w \lceil \log k \rceil$ -processor algorithm. Partition the jobs into $\lceil \log k \rceil$ groups, where the

³Recall that EDF-AC is the earliest deadline first algorithm with a simple form of admission control. See page 12 for definition.

i -th group contains all the jobs with value density in the range $[2^{i-1}, 2^i]$. Each group is given $2w$ processor executing MSA independently. Within each group, the value densities differ by at most a factor of $1 + \frac{k}{w}$, so the $2w$ processors match the value obtained by any off-line algorithm for jobs of this group. Therefore, the $2w \lceil \log k \rceil$ processors together can match the value obtained by any off-line algorithm for jobs with value densities in $[1, k]$. \square

Notice that w is a constant, and thus the algorithm is constant competitive when given $O(\log k)$ additional processors. For instance, when we take w as 2, the algorithm is 4 $\lceil \log k \rceil$ -processor 2-competitive.

The rest of this section is devoted in proving Theorem 4.4. For any input job sequence J , denote \mathcal{A}_J and \mathcal{O}_J as the set of jobs completed by MSA and an off-line algorithm, respectively. Consider any input job sequence \mathcal{I} . Partition \mathcal{I} into two subsets: those jobs that are completed by MSA (denoted \mathcal{S}), and those that are not completed by MSA (denoted \mathcal{F}). Recall that $\|\mathcal{S}\| = \sum_{J \in \mathcal{S}} v(J)$ for any job set \mathcal{S} .

Lemma 4.6 (The Lost Value Lemma [21]). *For some constant c , if $\|\mathcal{O}_{\mathcal{F}}\| \leq c \|\mathcal{A}_{\mathcal{I}}\|$, then $\|\mathcal{O}_{\mathcal{I}}\| \leq (c+1) \|\mathcal{A}_{\mathcal{I}}\|$.*

Proof. $\|\mathcal{O}_{\mathcal{I}}\| \leq \|\mathcal{O}_{\mathcal{F}}\| + \|\mathcal{O}_{\mathcal{S}}\| = \|\mathcal{O}_{\mathcal{F}}\| + \|\mathcal{A}_{\mathcal{S}}\| = \|\mathcal{O}_{\mathcal{F}}\| + \|\mathcal{A}_{\mathcal{I}}\| \leq (c+1) \|\mathcal{A}_{\mathcal{I}}\|$. \square

Lemma 4.7. *During the span of any job $J \in \mathcal{F}$, all Me 's are busy.*

Proof. Recall that a job in \mathcal{F} is not completed by MSA. Let $\ell(J) = d(J) - p(J)$. First we consider the period $[r(J), \ell(J)]$. At $r(J)$, no Me can admit J (otherwise J should be scheduled to completion on a Me). Therefore, all Me 's have already admitted enough jobs to keep them busy until after $\ell(J)$.

For the period $[\ell(J), d(J)]$, consider the instance when J is discarded. This happens when either it cannot find a Md to run at $\ell(J)$, or it has been preempted by another job with a later deadline (at line 22 in Algorithm 4.1). In both cases, all jobs in Md 's are with deadlines at least $d(J)$. During $[\ell(J), d(J)]$, when a Me completes all jobs it admitted, it switches its role with the Md of the same band. The new Me would be busy until at least $d(J)$. \square

Lemma 4.8. $\|\mathcal{O}_{\mathcal{F}}\| \leq \frac{k}{w} \|\mathcal{A}_{\mathcal{I}}\|$.

Proof. Recall that $\mathcal{O}_{\mathcal{F}}$ is the set of jobs completed by the off-line algorithm with input \mathcal{F} . By Lemma 4.7, whenever a job in \mathcal{F} can be scheduled, all the mw Me 's in MSA are busy. By the definition of EDF-AC, all jobs scheduled in Me 's are executed to completion. Therefore, all these jobs are in $\mathcal{A}_{\mathcal{I}}$. At any instance, for each job running by the off-line algorithm,

we can find w distinct jobs running in Me 's. The best the off-line algorithm can do is to schedule m jobs in $\mathcal{A}_{\mathcal{I}}$ each with value density k . Thus,

$$\frac{\|\mathcal{O}_{\mathcal{F}}\|}{k} \leq \frac{\|\mathcal{A}_{\mathcal{I}}\|}{w}. \quad \square$$

Proof of Theorem 4.4. The competitive ratio of MSA is

$$\frac{\|\mathcal{A}_{\mathcal{I}}\|}{\|\mathcal{O}_{\mathcal{I}}\|} \geq \frac{(1 + \frac{k}{w}) \|\mathcal{O}_{\mathcal{I}}\|}{\|\mathcal{O}_{\mathcal{I}}\|} = 1 + \frac{k}{w}. \quad \square$$

4.2 Additional Processors and Competitive Ratios

For uniprocessor scheduling, the algorithm SLACKER given by Kalyanasundaram and Pruhs [17] is speed- $(1 + 2\delta)(1 + \delta^{-1})(1 + \delta^{-1/2})(1 + \delta^{-1/2} + \delta^{-1})$ -competitive for any $\delta > 0$. Lam and To [25] gave an extension of SLACKER, denoted MSLACKER, which is speed- $(1 + 2\delta)(1 + 2\delta^{-1} + 4\delta^{-2})$ -competitive for multiprocessor scheduling.

In this section we show that MSLACKER has a natural extension for additional processors. Specifically, MSLACKER is a w -processor speed- $(1 + 2\delta)(1 + \frac{2}{w\delta} + \frac{4}{w\delta^2})$ -competitive algorithm, where $1 \leq w \leq 1 + 2\delta^{-1}$, for MFS- k . The improvement is more significant when the speedup is low. Figure 6 shows the competitiveness of MSLACKER with and without using additional processors. For example, with speed-2 processors, the original MSLACKER is 1-processor speed-2 21-competitive. With this extension, the competitive ratio can be improved to 2-processor speed-2 11-competitive, or 5-processor speed-2 5-competitive. The trade-off for the case of speed-2 processors is illustrated in Figure 7. This result should be contrasted with the $\Omega(\log k)$ speed-1 processor lower bound we have shown in the previous section.

MSLACKER is parameterized by two real values $\delta > 0$ and $c > 1$. MSLACKER is equipped with mw speed- s processors where $s = 1 + 2\delta$, and keeps an initially empty set of privileged jobs M . At any time, MSLACKER runs all jobs in M if M has mw or fewer jobs; otherwise, it runs the mw highest-value-density jobs in M . When a job J is released, J is added to M if M contains less than mw jobs, or $\rho(J) \geq c \cdot \rho(J_0)$ where J_0 is the mw -th highest-value-density job in M . If J cannot be added to M immediately, the same checking will be done to J again whenever a job is completed, up to time $d(J) - \delta \frac{p(J)}{s} - p(J)$. Notice that when a job completion occurs, if there are jobs other than J waiting to be added into M , we perform the checking for them in an arbitrary order. A job is removed from M if either it is completed, or it is certain to miss its deadline. Figure 8 shows how MSLACKER considers a job for execution.

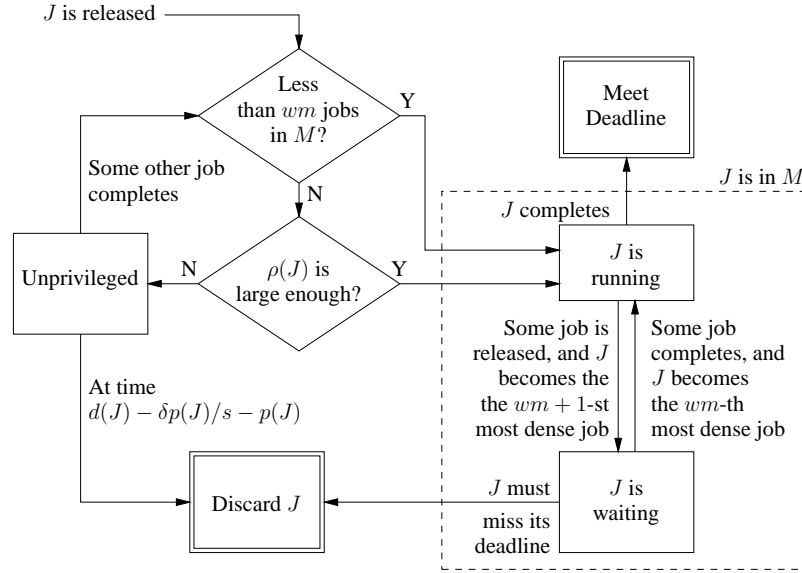


Figure 8: The life-cycle of a job J as scheduled by MSLACKER

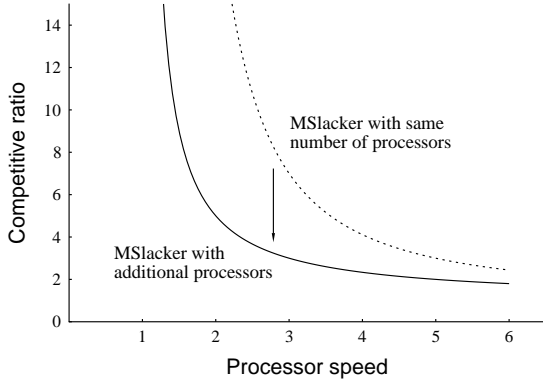


Figure 6: Competitiveness of MSLACKER with additional processors

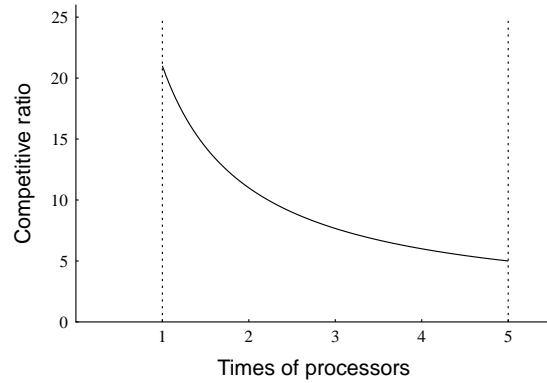


Figure 7: Trade-off between competitive ratio and times of processors for speed-2 processors.

Definition. Consider any input job sequence. Let \mathcal{O} be the set of jobs completed by an off-line algorithm, and let \mathcal{S} and \mathcal{R} denote the sets of jobs completed by MSLACKER and ever added into M respectively. By definition, $\|\mathcal{S}\| \leq \|\mathcal{R}\|$.

Definition. A job J , once released, is said to be *fresh*⁴ up to the time $d(J) - (1 + \delta)\frac{p(J)}{s}$.

⁴Notice that the definition of fresh here is different from the one in page 7.

The analysis is essentially the same as the original proof for MSLACKER. Intuitively, MSLACKER is very conservative and can complete most of the jobs added into M . We make use of the following fact that is proved in [25], which states that $\|\mathcal{S}\|$ is at least a significant fraction of $\|\mathcal{R}\|$.

Fact 4.9. (see Lemma 5.1 in [25]) $\|\mathcal{S}\| \geq \frac{\delta(c-1)-1}{\delta(c-1)} \|\mathcal{R}\|$.

We also show that the way MSLACKER selects the

jobs guarantees that $\|\mathcal{O}\|$ cannot exceed $\|\mathcal{R}\|$ too much (Lemma 4.11) by observing the following property.

Lemma 4.10. *At any time t when a job J has not yet completed by MSLACKER and J is still fresh, then either MSLACKER is executing J , or MSLACKER is executing mw other jobs each with value density at least $\rho(J)/c$.*

Proof. At time t , J may or may not be in M . If J is in M and not being executed, then the value density of any job MSLACKER is executing is at least $\rho(J)$. Next, we consider J not in M . Throughout the period $[r(J), t]$, J is not qualified to get into M , i.e., M contains at least mw jobs, and $\rho(J) < c \cdot \rho(J_0)$ where J_0 is the mw -th highest-value-density job in M . At time t , MSLACKER is executing m other jobs each with value density at least $\rho(J)/c$. \square

We are now ready to show the upper bound of $\|\mathcal{O}\|$.

Lemma 4.11. *For $1 \leq w \leq c$, $\|\mathcal{O}\| \leq \|\mathcal{S}\| + \frac{c}{w\delta} \|\mathcal{R}\|$.*

Proof. Partition \mathcal{O} into $\mathcal{O}^c = \mathcal{O} \cap \mathcal{S}$ and $\mathcal{O}^u = \mathcal{O} - \mathcal{S}$. Since $\mathcal{O}^c \subseteq \mathcal{S}$, $\|\mathcal{O}\| \leq \|\mathcal{S}\| + \|\mathcal{O}^u\|$. Intuitively, each job in \mathcal{O}^u must be fresh for a large proportion of the time when off-line chooses the job for execution. On the other hand, Lemma 4.10 guarantee that at such time MSLACKER must have chosen jobs with large value density for execution. Details are as follows.

For any job J in \mathcal{O}^u , define $a_1(J)$ (respectively $a_2(J)$) to be the total amount of time when the adversary executes J while J is fresh (respectively J is no longer fresh). By definition, $a_1(J) + a_2(J) = p(J)$. For any job $J \in \mathcal{O}^u$, $a_2(J) \leq (1 + \delta) \frac{p(J)}{s}$, and $a_1(J) = p(J) - a_2(J) \geq \delta \frac{p(J)}{s}$. Thus, $\frac{\delta}{s} v(J) = \frac{\delta}{s} p(J) \rho(J) \leq a_1(J) \rho(J)$, and $\frac{\delta}{s} \|\mathcal{O}^u\| \leq \sum_{J \in \mathcal{O}^u} a_1(J) \rho(J)$.

To derive an upper bound of $\|\mathcal{O}\|$, we consider $a_1(J)$ for each job $J \in \mathcal{O}^u$. By definition, every job $J \in \mathcal{O}^u$ is not completed by MSLACKER. At any time when the adversary executes a job $J \in \mathcal{O}^u$ while J is fresh, Lemma 4.10 tells us that MSLACKER either executes J , or executes mw jobs each of value density at least $\rho(J)/c$. In general, at any time t , let X_t be the set of fresh jobs in \mathcal{O}^u currently executed by the adversary; then for each job $J \in X_t$, either it is running by MSLACKER, or we can identify w distinct jobs currently executed by MSLACKER with job density at least $\rho(J)/c$. In other words, the total value density of jobs in X_t is at most c/w times the total value density of jobs currently executed by MSLACKER. To bound $\sum_{J \in \mathcal{O}^u} a_1(J) \rho(J)$, it suffices to consider the sum over all time t of the total value density of jobs in X_t , which is at most c/w times of the sum over all time t of the total value density of jobs executed by

MSLACKER at time t . Note that each job $J \in \mathcal{R}$ can contribute a quantity of at most $\rho(J) \frac{p(J)}{s}$ to the latter sum. Thus, the latter sum can be expressed as $\sum_{J \in \mathcal{R}} \rho(J) \frac{p(J)}{s}$, which is equal to $\frac{1}{s} \|\mathcal{R}\|$. In summary, $\sum_{J \in \mathcal{O}^u} a_1(J) \rho(J) \leq \frac{c}{ws} \|\mathcal{R}\|$. Therefore, $\delta \|\mathcal{O}^u\| \leq \frac{c}{w} \|\mathcal{R}\|$, and $\|\mathcal{O}\| \leq \|\mathcal{S}\| + \|\mathcal{O}^u\| \leq \|\mathcal{S}\| + \frac{c}{w\delta} \|\mathcal{R}\|$. \square

Theorem 4.12. *MSLACKER is w -processor speed- $(1 + 2\delta)(1 + \frac{2}{w\delta} + \frac{4}{w\delta^2})$ -competitive, where $1 \leq w \leq 1 + 2\delta^{-1}$, when c is chosen as $1 + 2\delta^{-1}$, for MFS- k .*

Proof. It follows from Fact 4.9 and Lemma 4.11. \square

Corollary 4.13. *For any real $s > 1$, MSLACKER is $(1 + \frac{4}{s-1})$ -processor speed- $s(1 + \frac{4}{s-1})$ -competitive for MFS- k .*

4.3 Concluding Remarks

We have showed that for MFS- k , $\Theta(\log k)$ additional processors are required to attain a constant competitive ratio. We have also showed how to use additional processors to improve the competitive ratio of MSLACKER.

5 Conclusion

5.1 Summary

In this thesis, we have discussed the use of additional resources, namely, faster and additional processors, for the on-line firm deadline scheduling problem. We considered this problem under different settings: uniprocessor or multiprocessor scheduling, uniform or general value density, and optimal or constant competitive algorithms. Table 4 and 5 below summarize the up-to-date results. Results marked with the a cross (\dagger) are shown in this thesis.

Notice that for competitive results (Table 5), only the order of magnitude is given. This is because the absolute value of resource requirement can be tuned according to the desired competitive ratio.

In addition, we have also shown the following two results:

- A speed- $O(1)$ optimal algorithm for UFS- k when all jobs are tight.
- The trade-off between additional processors and competitive ratio for multiprocessor scheduling.

5.2 Discussion

5.2.1 Speed versus processors

Intuitively, one speed- x processor and x speed-1 processors offer similar power — to allow x units of work to be done in one unit of time. Yet the results show that speed is much more powerful; indeed, to achieve constant competitiveness for general value density, $O(1)$ times faster processors are sufficient, but not for any w -processor algorithm unless $w = \Omega(\log k)$.

In fact, the major difference between speed and processors lies on their power to “correct mistakes”, i.e., what the on-line algorithm can do once it realizes that it has not scheduled jobs that will be completed by the optimal off-line algorithm. When faster processors are used, the on-line algorithm can catch up the optimal off-line algorithm by scheduling those jobs with its extra speed. However, this is not feasible when using additional processors. Algorithms with additional processors can only schedule more jobs, hoping that the value it obtains from those jobs is enough to compensate for any mistakes in choosing the jobs. When $k = 1$, jobs are equally important and the difference is not that significant. This is not true for general value density, where quality outbids quantity. This explains the $O(\log k)$ requirement for general k .

5.2.2 Optimality with a faster processor

For UFS-1, the speed requirement for achieving optimality in the uniform-value-density setting is in the range $[\phi, 2]$. We conjecture that the lower bound ($\phi \approx 1.618$) is the real bound, i.e., there exists a speed- ϕ optimal algorithm. However, the new algorithm would be very different from EDF-AC as we conjecture that the new algorithm must be able to discard jobs it has previously scheduled.

For UFS- k , the speed requirement for optimality is in the range $[2, 4 \lceil \log k \rceil]$. The upper bound is very different from that for constant competitiveness, where speed- $O(1)$ is sufficient for the latter. On the other hand, with the tight jobs assumption, a speed- $O(1)$ optimal algorithm is found. We believe that tight jobs are the most difficult cases, as most known lower bound results are all based on tight jobs only. This suggests that there is room for improvement for the upper bound. Indeed, we conjecture that the upper bound can also be improved to $O(1)$.

5.2.3 Optimality in multiprocessor scheduling

No algorithm is known to be optimal using additional speed-1 processors for multiprocessor scheduling. The major difficulty in providing such guarantee is on bounding the total value obtained by an off-line algorithm. The optimal schedule changes dramatically upon the introduction of an additional job, which makes comparisons between on-line and off-line schedules intractable. By making rough approximations, it is possible to come up with competitive algorithms. Yet a more precise analysis is required for any optimality result.

Moreover, little is known about the co-operations between processors. In fact, most of the algorithms simply partition the jobs according to their value density and assign them to different sets of processors, instead of using the processors as complement of each other. A better understanding of multiprocessor scheduling is the key to solve this problem.

5.3 Open Problems

This work leads to a few possible research directions:

- No result is known for achieving optimality using only additional processors for multiprocessor scheduling. What guarantees can be given when only additional processors are available?
- We have assumed that migration is allowed with no cost. Recent works (e.g. [16, 20]), however, considered schedules that do not use migration. Can similar performance guarantees be made without migration?

	Resource Type	$k = 1$	$k \geq 2$
$m = 1$	Processors	2^\dagger	$\Theta(\log k)^\dagger$
	Speed	$1.618^\dagger [23] \leq s \leq 2 [25]$	$2^\dagger \leq s \leq 4 \lceil \log k \rceil [25]$
$m \geq 2$	Processors	?	?
	Speed	$s \leq 3 [25]$	$s \leq 4 \lceil \log k \rceil [25]$

Table 4: Additional resource requirement for achieving optimality

	Resource Type	$k = 1$	$k \geq 2$
$m = 1$	Processors	1^\natural	$\Theta(\log k)^\dagger$
	Speed	1^*	$O(1) [17]$
$m \geq 2$	Processors	$O(1) [21]$	$\Theta(\log k)^\dagger$
	Speed	$O(1)^{**}$	$O(1) [25]$

Table 5: Additional resource requirement for achieving constant competitiveness

- Can matching bounds for achieving optimality with a faster processor for uniprocessor systems be found? In particular, is it possible to find speed- ϕ and speed- $O(1)$ optimal algorithms for uniform and general value density, respectively?
- The speed requirement, even with a constant speed factor, may not always be feasible. Given the $\Omega(\log k)$ lower bounds for additional processors in many settings, a more realistic approach is to use more slightly faster processors instead. Precisely, given a speed factor s slightly larger than 1, how many speed- s processors are sufficient to guarantee optimality? The trade-off in Section 4.2 answers part of this question.
- Can the upper bounds be improved by making use of randomization?
- What guarantees can be given for other objectives, for example maximizing job completions or minimizing response time?
- Most lower bound results make use of tight jobs. If there is guarantee on the “tightness” of the jobs (e.g., the ratio of the span to the processing time), can we find algorithms using less additional resources?

*No additional resources is needed to achieve constant competitiveness; indeed, D^{over} is 4-competitive without using any additional resource [22].

‡EDF-AC, using a speed-3 processor, is already 1-competitive for MFS-1 [25].

References

- [1] B. Awerbuch, Y. Azar, S. Leonardi, and O. Regev. Minimizing the flow time without migration. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 198–205, 1999.
- [2] S. Baruah. Overload tolerance for single-processor workloads. In *IEEE Symposium on Real time technology and application*, pages 2–11, 1998.
- [3] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang. On the competitiveness of on-line task real-time task scheduling. *Journal of Real-Time Systems*, 4(2):125–144, 1992.
- [4] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier, and D. Shasha. On-line scheduling in the presence of overload. In *Proceedings of the IEEE Thirty-second Annual Symposium on the Foundations of Computer Science*, pages 101–110, 1991.
- [5] Sanjoy K. Baruah, Jayant Haritsa, and Nitin Sharma. On-line scheduling to maximize task completions. In *Proceedings of the Real-Time Systems Symposium*, pages 228–236, 1994.
- [6] P. Berman and C. Coulston. Speed is more powerful than clairvoyance. *Nordic Journal of Computing*, 6(2):181–193, Summer 1999.
- [7] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.
- [8] Bhaskar DasGupta and Michael A. Palis. Online real-time preemptive scheduling of jobs with deadlines. In *Proceedings of the Third International Workshop on Approximation Algorithms for Combinatorial Optimization*, volume 1913 of *Lecture Notes in Computer Science*, pages 96–107. Springer, September 2000.
- [9] Michael L. Dertouzos. Control robotics: the procedural control of physical processes. In *Proc. IFIP Congress*, pages 807–813, 1974.
- [10] Michael L. Dertouzos and Aloysius Ka-Lau Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, December 1989.
- [11] Jeff Edmonds. Scheduling in the dark. *Theoretical Computer Science*, 235(1):109–141, March 2000.
- [12] T.F. Gonzales and D.B. Johnson. A new algorithm for preemptive scheduling of trees. *Journal of the ACM*, 27:287–312, 1980.
- [13] K.S. Hong and J.Y.-T. Leung. On-line scheduling of real-time tasks. *IEEE Transactions on Computers*, 41(10):1326–1331, 1992.
- [14] Bala Kalyanasundaram and Kirk Pruhs. Minimizing flow time nonclairvoyantly. In *Proceedings of the Thirty-eighth Annual Symposium on Foundations of Computer Science*, pages 345–352, 1997.
- [15] Bala Kalyanasundaram and Kirk Pruhs. Maximizing job completions online. In *Proceedings of the Sixth European Symposium on Algorithms*, pages 235–246, 1998.
- [16] Bala Kalyanasundaram and Kirk Pruhs. Eliminating migration in multi-processor scheduling. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 499–506, 1999.
- [17] Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM*, 47(4):617–643, July 2000.
- [18] C.Y. Koo, T.W. Lam, T.W. Ngan, and K.K. To. On-line scheduling with tight deadlines. In *Proceedings of the Twenty-sixth International Symposium on Mathematical Foundations of Computer Science*, 2001. To appear.
- [19] G. Koren. *Competitive On-line Scheduling for Overloaded Real-time Systems*. PhD thesis, New York University, 1993.
- [20] Gilad Koren, Amihod Amir, and Emanuel Dar. The power of migration in multi-processor scheduling of real-time systems. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 226–235, 1998.
- [21] Gilad Koren and Dennis Shasha. MOCA: A multiprocessor on-line competitive algorithm for real-time system scheduling. *Theoretical Computer Science*, 128(1–2):75–97, June 1994.
- [22] Gilad Koren and Dennis Shasha. D^{over} : An optimal on-line scheduling algorithm for overloaded real-time systems. *SIAM Journal of Computing*, 24(2):318–339, April 1995.
- [23] Tak-Wah Lam, Tsuen-Wan Ngan, and Kar-Keung To. On the speed requirement for optimal deadline scheduling in overloaded systems. In *Proceedings of the Fifteenth International Parallel and Distributed Processing Symposium*, page 202, 2001.

- [24] Tak-Wah Lam and Kar-Keung To. Trade-offs between speed and processor in hard-deadline scheduling. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 623–632, 1999.
- [25] Tak-Wah Lam and Kar-Keung To. Performance guarantee for online deadline scheduling in the presence of overload. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 755–764, 2001.
- [26] Cynthia A. Phillips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 140–149, 1997.
- [27] Jiří Sgall. On-line scheduling — a survey. In A. Fiat and G. Woeginger, editors, *On-line Algorithms: The State of the Art*, pages 196–231. Springer Verlag, 1998.
- [28] D.B. Shmoys, J. Wein, and D.P. Williamson. Scheduling parallel machines on-line. *SIAM Journal of Computing*, 24:1313–1331, 1995.
- [29] J.A. Stankovic, M. Spuri, K. Ramamritham, and G.C. Buttazzo. *Deadline scheduling for real-time systems: EDF and related algorithms*. Kulwer Academic Publishers, 1998.