Energy-efficient Scheduling Algorithms

The Cost of High Performance

Peter Kling

University of Paderborn

Reviewers:

- Prof. Dr. Friedhelm Meyer auf der Heide, University of Paderborn
- Prof. Dr. Kirk Pruhs, University of Pittsburgh
- Prof. Dr. Christian Scheideler, University of Paderborn

To Nadine, my beloved wife.

Contents

List of Theorems

List of Figures

List of Listings

Preface

Writing this thesis to get a Ph.D. degree was by far the biggest project I ever approached. Writing the following lines is by far the most satisfying part of it, as it signals myself that I'm done. If not for the support of many colleagues, friends, and my family, I would not be in the position to savor this moment. I am deeply grateful to all of you.

There are several people who had a substantial influence on my work and who deserve more personal words of gratitude. One of them is my mentor and supervisor Friedhelm Meyer auf der Heide. He not only gave me the possibility to earn a Ph.D. degree but managed a perfect balance between guiding me and letting me develop my own research profile. Without your advice and trust, this thesis would not exist. Thank you.

Special thanks go to Kirk Pruhs, who invited me to work with him and his research group, supported me in my postdoctoral plans, and agreed to review this thesis. I truly enjoyed the work with you, your postdocs, and your Ph.D. students. Thank you.

Working on my Ph.D. thesis was a time of many ups and downs. But even when algorithms failed, proofs fell apart, and equations did not yield, my colleagues ensured that work never became a burden. Andreas, with whom I shared my office almost right from the start, made the early hours at work less lonely and seemed to have an endless stock of ideas for decorations, culminating in our very own, full-grown office palm tree. Sören ("dance-for-no-reason") was as musical an office-mate as I could hope for, and our scheduling discussions and research sessions were always extremely productive, not least because of his "Gegenbeispiel GmbH". A very special thank you goes to my namesake Peter ("der Freundliche"). I don't think there is anyone with whom I had more discussions, be it on primal-dual algorithms, Starcraft 2 and eSports, or everyday life. To all my other colleagues, I beg pardon for not naming you explicitly. I enjoyed playing soccer in our weekly *Theory Soccer Sessions*, spending my lunch breaks with you playing *World of Padman*, and chatting during our regular meetings at the coffee machine. Thank you all.

I owe much to my family, especially to my parents Thomas and Liesel. You paved the way for my education, helped me in countless ways during my studies, and always provided a safe haven. Thank you.

Most of all, I thank the one person who had and has to endure all my moods, complaints, practice talks, and long working hours. She who (literally) uses brute force to ensure that I do not neglect other (more?) fun and less analytical parts of life. She who is always there for me. She to whom this thesis is dedicated. *Thank You!*

CHAPTER 1

Introduction

What matters most to the computer designers at Google is not speed, but power, low power, because data centers can consume as much electricity as a city.

Eric Schmidt, former CEO Google [ML02]

High performance comes at a high cost, or so it seems if one considers technology operating at extremes. The fuel consumption of sport cars, the cooling cost of today's data centers, or the power consumption of high-end graphics cards – all of these are immense, and doubling the respective performance easily increases the cost by a factor of eight, nine, or even more. Thus, it is not surprising that there is considerable research effort in improving the efficiency of technical systems throughout all phases of their design and development process. In this thesis, I study methods that deal with such efficiency issues on an algorithmic level. In particular, I design theoretic models that isolate specific aspects of a problem, with the goal to study their inherent complexity and to design provably optimal as well as provably good approximation and online algorithms.

A distinctive feature of my work is that it considers energy as a major goal in algorithm design. Classical efficiency and complexity measures study merely time and space as the primary computational resources. This is not too surprising, as these are probably the most obvious and perceivable quality measures. However, considering recent technical developments, a new trend becomes apparent: energy claims its position as a first-class citizen among quality measures. Apart from the increased amount of research under the catchphrase *green computing*, one can experience this phenomenon much more directly:

- battery life has become a major selling point for mobile devices like notebooks, e-book readers, and (especially) mobile phones [Ste13],
- data center operators like Apple, Facebook, and Google aggressively advertise their green computing efforts [App13a; Fac13; Goo13],
- and even mainstream operating systems like OS X 10.9 (a.k.a. Mavericks) focus on energy efficiency [App13b].

Algorithmic research must live up to this increased public awareness and rapid development of energy efficiency. Any software engineer learns how to compute the time and space complexity of algorithms, e.g., by using tools like the big O notation. However, we still lack a sound understanding and scientific framework to reason about energy in a similar, systematic way. How can we come up with such a framework, given that energy behaves physically so differently than time and space? This is the driving research question in energy-aware algorithm design. With my thesis, I hope to provide some insights that get us closer to an answer.

1.1 A Primer to Speed Scaling

In many applications, the need for extreme operating speeds may occur once in a while, but for the majority of time a lower performance level suffices to handle the task at hand. In the case of computer systems, chip manufacturers like Intel and AMD exploit this fact by providing means to throttle a processor's speed. Such techniques are often referred to as dynamic voltage scaling or – the term used in my thesis – *speed scaling*. Speed scaling allows to adjust a processor's speed by using one of several (discrete) speed levels, where lower levels consume less energy. If the workload becomes too high to guarantee a sufficient quality of service, a processor may temporarily be set to a higher speed level. Theoretical research in this area was initiated by the seminal work of Yao et al. [YDS95]. The authors combined classical deadline scheduling on one processor with a model for speed scaling, striving to schedule all jobs while minimizing the total energy consumption.

More precisely, Yao et al.'s speed scaling model considers a single processor that can run at an arbitrary, continuous speed and has to schedule a set of n jobs. Each job j has a release time r_j, a deadline d_j, and a processing volume p_j. The processor can run at any speed $s \geq 0$ and consumes $\mathrm{P}(s) = s^\alpha$ units of energy per time unit while doing so. Here, $\alpha > 1$ is a constant often referred to as the *energy exponent*. In retrospect, one of the most important results in [YDS95] was the design of the online algorithm *Optimal Available* (OA), which was later proven to be exactly α^α-competitive by Bansal et al. [BKP04]. The best known lower bound for deterministic algorithms is $\frac{e^{\alpha-1}}{\alpha}$ [Ban+09]. Due to its simplicity, OA is used as the basis of many sophisticated algorithms both in the original speed scaling model as well as in its numerous variants.

1.2 Thesis Overview

My thesis aligns with the rich body of literature on variants of Yao et al.'s speed scaling model. It consists of four parts, the first three of which are directly related to this model. Parts one (Chapter 3) and two (Chapter 4) give results for different price-collecting variants of the original problem, where the scheduler may miss a job's deadline if it pays a job-specific penalty. Part three (Chapter 5) replaces the deadline requirement by the classical flow time objective. The last part (Chapter 6) introduces a new type of resource constrained scheduling, in the remainder denoted as *shared resource scaling*. This last part does not directly consider energy, but it might be a first step towards a theoretical model where energy is seen as a resource shared by several processors. The remainder of this section provides a short description of each part and the corresponding results.

Profitable Deadline Scheduling This part generalizes and improves work of Chan et al. [CLL10a]. They consider the price-collecting version of the deadline speed scaling model for a single processor. That is, the scheduler may miss deadlines if it is willing to pay a job-specific penalty corresponding to the jobs' *values* v_j. Chan et al. [CLL10a] design an algorithm based on Optimal

Available and adapt the potential function analysis by Bansal et al. [BKP04]. They prove a competitiveness of at most $\alpha^\alpha + 2e\alpha$. I improve upon this in two respects:

(a) I design an algorithm with a competitiveness of exactly α^α.

(b) I generalize the algorithm and its analysis to arbitrary many processors while maintaining the competitive ratio of α^α.

The proposed algorithm has similarities to Chan et al.'s, but its design and analysis is significantly different from prior approaches and from the typical potential function argument (which is dominant in this area). It builds upon and extends ideas of Gupta et al. [GKP12], utilizing techniques from convex optimization. The systematic nature and improved results give hope that this approach might turn out useful in the analysis of other speed scaling problems.

The model, analysis, and results presented in this part are based on the following publication:

> **2013** (with P. Pietrzyk). "Profitable Scheduling on Multiple Speed-Scalable Processors". In: *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, cf. [KP13].

Slow Down & Sleep for Profit The second part extends the results of Chan et al. [CLL10a] in another way, by considering the more realistic processor model of Han et al. [Han+10]. Here, the processor's energy consumption at speed s is given by $\mathrm{P}(s) = s^\alpha + \beta$, where $\beta > 0$ models the energy necessary to keep the system awake (e.g., to maintain the registers). The scheduler may put the processor into a *sleep state*, where energy consumption is negligible but resuming to work causes additional costs $\gamma > 0$. Most modern devices feature such a technique (e.g., hibernation). The idea extends to data centers, which may power off some systems during times of low load but need to reactivate them fast in case of sudden workload peaks.

Han et al. [Han+10] give an $\alpha^\alpha + 2$-competitive algorithm for the online setting without price-collection. My work shows that combining price-collection with the more realistic processor model adds a new kind of complexity. More specifically, a natural class of "local" algorithms has – in contrast to the setting without sleep states – an unbounded competitive ratio. The maximum *value* *density* $\delta_{\max} = \max_j \frac{v_j}{p_j}$ (the maximum ratio between a job's value and its total processing volume) turns out to be inherently connected to the achievable competitive ratio, as shown by the two main results:

(a) The competitiveness of any such algorithm is lower-bounded by $\Omega(\delta_{\max})$.

(b) Such an algorithm can achieve a competitiveness of $\alpha^{\alpha} + 2e\alpha + O(\delta_{\max})$.

The constants hidden by the Landau symbols match. These results extend to models with a bounded maximum speed.

The model, analysis, and results presented in this part are based on the following publication:

> **2012** (with A. Cord-Landwehr and F. Mallmann-Trenn). "Slow Down and Sleep for Profit in Online Deadline Scheduling". In: *Proceedings of the 1st Mediterranean Conference on Algorithms (MedAlg)*, cf. [CKM12].

Trading Energy for Responsiveness In contrast to the previous parts, this part considers an offline problem. Moreover, jobs are no longer assumed to have deadlines. Instead, the classical scheduling objective of (fractional) weighted flow time (a.k.a. response time) plus energy is considered. This problem's complexity is a long-standing open question. Pruhs et al. [PUW04] prove that, for non-fractional flow, it can be solved optimally by a polynomial-time algorithm if all jobs have size one. Cole et al. [Col+12] consider the problem with general job sizes but for fractional flow. While they cannot settle the problem's complexity, they give an algorithm that recognizes an optimal schedule in polynomial time. The results presented in my thesis make a further step towards settling the problem's complexity by giving a polynomial-time algorithm for the fractional flow version if there is only a discrete number of speeds. Note that the algorithm's running time is polynomial in the number of both jobs and speeds. Its design uses a geometric approach based on structural properties obtained from the problem's primal-dual formulation.

The model, analysis, and results presented in this part are based on the following publication:

> **2014** (with A. Antoniadis, N. Barcelo, M. Consuegra, M. Nugent, K. Pruhs and M. Scquizzato). "Efficient Computation of Optimal

Energy and Fractional Weighted Flow Trade-off Schedules". In: *Proceedings of the 31st Symposium on Theoretical Aspects of Computer Science (STACS)*. in press, cf. [Ant+14].

Sharing Scalable Resources In the last part, not energy efficiency but resource constrained scheduling is considered. In contrast to most classical resource constrained scheduling models, the proposed model allows the scheduler to partially process a job if only a fraction of the requested resource is available. As a simple example, consider jobs with extensive I/O transactions and a number of processors sharing a single I/O bus of a fixed bandwidth. Granting a job its full requested bandwidth guarantees minimal processing time. However, even if granted only half of its requested bandwidth the job can be processed, if only at half the speed. That is, the speed a job is processed with depends on the amount of resource assigned to it.

The idea of such resource dependent processing speeds is not completely new. Józefowska and Węglarz [JW98] consider so called *discrete-continuous* scheduling problems. The "discrete" part of the term refers to normal multiprocessor scheduling, the processors constituting the discrete resource. The "continuous" part relates the processing speed of a job j to the share it receives of the (continuous) resource. In general, results for the discrete-continuous scheduling model seem to be either very pessimistic or heuristic in nature. To approach this problem from a more analytical angle, my thesis focuses on the continuous aspect (avoiding additional complexity originating from normal scheduling decisions). To this end, I consider a simplified model with discrete time steps and assume that jobs are already assigned to one of the m processors and have a predefined ordering. Thus, the scheduler only has to assign the resource each processor is granted at a time step. For instances containing only unit size jobs, I prove NP-hardness in m and present an approximation algorithm with a worst-case approximation ratio of exactly $2 - 1/m$.

The model, analysis, and results presented in this part are based on the following technical report. An extended version of these results is currently in preparation for submission.

2014 (with F. Meyer auf der Heide, L. Nagel, S. Riechers and T. Süß). "Sharing Scalable Resources". In preparation, cf. [Kli+14].

CHAPTER 2

Preliminaries

Give me six hours to chop down a tree and I will spend the first four sharpening the ax.

Abraham Lincoln

Before we head into the research parts of this thesis, let us prepare ourselves by introducing a few preliminaries that ease the further discussions. Note that all the Chapters 3, 4, 5, and 6 are, in principle, self-contained. In particular, they are laid out such that readers with a working knowledge of theoretical computer science can pick an arbitrary chapter and follow the line of thought without further ado, albeit not necessarily at full detail.

Field Manual This chapter is not exactly intended to be read front to back. Instead, the reader is encouraged to cherry-pick what raises her interest and what seems most beneficial. For readers not familiar with the speed scaling terminology, I recommend to read at least Section 2.2, which introduces a basic model that will re-appear in different variations throughout all remaining chapters.

2.1 Basics: Approximation & Online Algorithms

In the following we recapitulate the most basic definitions and intuitions for approximation and online algorithms. In particular, we consider how to measure an algorithm's quality by means of its *approximation ratio* and *competitive ratio*. For excellent, much more elaborate introductions to these topics, see the books by Vazirani [Vaz01] and Borodin and El-Yaniv [BE98], respectively.

Approximation Algorithms Given that certain complexity assumptions are true, computation of optimal solutions is often provably intractable. This is the case for so called NP-hard problems, which cannot be solved in time that depends polynomially on the input size if the complexity classes NP and P are not equal (the infamous NP vs. P problem). This inequality seems quite likely to most experts; in fact, most cryptographic primitives are based on this or similar assumptions. Since many real-world problems (e.g., flight scheduling, project planning, manufacturing processes, ...) fall into the class of NP-hard problems and must be solved regularly and efficiently, we need alternatives to computing optimal solutions. Approximation algorithms form one such alternative. The typical goal is to design efficient (i.e., running in time that is polynomial in the input) algorithms that compute nearly optimal (i.e., provably close to optimal) solutions. There are different ways to measure the quality of approximation algorithms. The most well-known is probably via the (relative) worst-case approximation ratio (which is also the quality measure used in this thesis). For minimization problems, one can define it as follows:

Definition 2.1 (Worst-case Approximation Ratio)**.** Let Π denote the class of all feasible instances for a given minimization problem. Moreover, let $A(I)$ and $\mathrm{OPT}(I)$ denote the cost of the approximation algorithm A and an optimal solution for a given problem instance I. Then the *worst-case approximation ratio* of A is

$$\gamma := \sup_{I \in \Pi} \frac{A(I)}{\mathrm{OPT}(I)}. \tag{2.1}$$

Note that $\gamma \geq 1$ for any algorithm A (and $\gamma = 1$ only for optimal algorithms).

Online Algorithms Approximation algorithms essentially trade quality for computation time. But sometimes other resources than computation time are restricted. One restriction in many real-world scenarios is the lack of knowledge, especially with respect to the future. This is often referred to as the *online setting* of a problem. Scheduling is a typical example for an area where online problems occur: when jobs arrive over time, the scheduler can often not know how many and what jobs are yet to come. Graham [Gra66] was probably among the first to study such scenarios (albeit not in the terminology of online problems). He considered the problem of how to schedule n jobs with different processing volumes on m identical machines to minimize the latest completion time. In particular, he studied the greedy algorithm that considers jobs in a fixed order and assigns the current job to the machine of smallest load. He proved that its quality varies by a factor of at most $2 - 1/m$, depending on the job order. This is a valid online algorithm: it needs no knowledge of future jobs but takes decisions only based on the past and current jobs. It follows easily from [Gra66] that this greedy algorithm is by a factor of at most $2 - 1/m$ worse than an optimal (offline) algorithm.

This kind of statement, bounding the worst-case quality of an algorithm relative to an optimal solution, is probably the most common quality measure used for online algorithms and is referred to as the algorithm's (worst-case) *competitive ratio*. Formally, it is simply the worst-case approximation ratio of a given (deterministic) online algorithm. See [BE98] or, more compact and focused on scheduling problems, [PST04] for an in-depth introduction to online algorithms.

2.2 The First Speed Scaling Model

Let us give a formal pendant to the speed scaling description from Section 1.1. Remember that speed scaling describes a technique to adapt a processor's speed at runtime to the current workload. The model we present is due to the seminal work of Yao et al. [YDS95], which initiated theoretical research in this area. We adapt the notation to mimic (as far as possible) the notation of later chapters.

2.2.1 Model Notions

Processor Model Consider a single speed-scalable *processor*. It can run at any speed $s \in \mathbb{R}_{\geq 0}$. While doing so it processes work at a rate of s and consumes energy at a rate of $\mathrm{P}(s)$. Here, $\mathrm{P}\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ is a typically continuous and convex function, often referred to as the processor's *power function*. If the speed at time t is given by a *speed function* $s\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, the workload processed and energy consumed during a time interval $[t_1, t_2)$ can be expressed by

$$W(t_1, t_2) := \int_{t_1}^{t_2} s(t)\,\mathrm{d}t \tag{2.2}$$

and

$$E(t_1, t_2) := \int_{t_1}^{t_2} \mathrm{P}(s(t))\,\mathrm{d}t, \tag{2.3}$$

respectively.

Typically, the power function is assumed to be of the form $\mathrm{P}(s) = s^{\alpha}$, with a constant α that is roughly in the range two to five. The standard example for this *energy exponent* is $\alpha = 3$, which leads to a model that mimics the energy consumption of CMOS-based systems reasonably well[1].

Job Model The processor has to process a set of *n jobs* $\mathcal{J} = \{1, 2, \dots, n\}$. Each job j has a *release time* r_j, a *deadline* d_j, and a *processing volume*[2] p_j. A job must be fully processed between its release time and its deadline. Jobs are preemptable (i.e., can be suspended at any time and resumed later on) but cannot be processed in parallel (i.e., the processor can process at most one job at any time).

Schedules A *schedule* S assigns jobs to the processor, ensuring that at most one job is processed at any time, and sets the processor's speed. It can be described by a speed function $s\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ and a job assignment function $J\colon \mathbb{R}_{\geq 0} \to \mathcal{J}$. We say a schedule is feasible if it fully processes all jobs (between

[1]This is typically referred to as the *cube-root rule* [Bro+00].
[2]We will refer to a job's processing volume also as its size, workload, or simply work.

their release times and deadlines). That is, feasible schedules must ensure that

$$\sum_{\substack{[t_1,t_2)\subseteq J^{-1}(j)\cap[r_j,d_j)\\ \text{maximal}}} W(t_1,t_2) \geq p_j \tag{2.4}$$

holds for all $j \in \mathcal{J}$. The total energy consumption of S is $E_S := E(0,\infty)$.

Schedule Quality The quality of a schedule is measured by its total energy consumption. We strive for schedules that are as energy-efficient as possible. Thus, our general objective is to find a schedule that minimizes E_S.

We consider essentially two different settings for this quality measure: the classical *offline* setting and the *online* setting. As described in Section 2.1, these settings differ in that an algorithm in the online setting has no a priori knowledge of the jobs and their properties, while an offline algorithm knows the number of jobs as well as all their release times, deadlines, and processing volumes right from the start.

2.2.2 Optimal Offline Algorithm & Optimal Available

Besides introducing this model, Yao et al. [YDS95] gave an efficient optimal offline algorithm, typically referred to as *YDS*. In essence, YDS is a greedy algorithm. It iteratively computes intervals of maximum density, where the density of an interval $[t_1,t_2)$ is defined by

$$\delta(t_1,t_2) := \frac{\sum_{t_1 \leq r_j, d_j \leq t_2} p_j}{t_2 - t_1}. \tag{2.5}$$

It is easy to see that a schedule running at speed $\delta(t_1,t_2)$ during the interval $[t_1,t_2)$ and scheduling jobs by earliest deadline first finishes all corresponding jobs. The YDS algorithm proceeds by deleting the scheduled jobs and the corresponding time interval from the given input (adjusting release times and deadlines $\geq t_1$). We will not elaborate on the correctness and running time of this algorithm, but refer instead to [BKP07]. Nevertheless, note that Bansal et al. [BKP07] gave a very elegant optimality proof of YDS, based on a convex programming formulation and the corresponding KKT conditions (see Chapter 3 for further details).

Optimal Available For the online setting, Yao et al. [YDS95] proposed an algorithm called *Optimal Available* (OA). Its algorithmic idea is rather simple and lives up to the name: whenever a new job is released, compute an optimal schedule for the available (remaining) workload (e.g., via YDS). The authors did non provide an analysis of OA's competitive ratio, and it took nearly ten years until such an analysis was provided by Bansal et al. [BKP04]. There, the authors used a sophisticated potential function argument to prove that OA is exactly α^{α}-competitive. Apart from [YDS95], this is probably one of the most influential papers in the speed scaling literature, as the proposed potential function method has been applied in a multitude of related speed scaling problems. One example is Chapter 4 of this thesis, where we adapt this analysis to a price collecting variant of speed scaling featuring a sleep state. See Section 2.3 for further examples.

A noteworthy structural property of OA follows easily from the observation that all the intermediate schedules computed when a new job is released correspond to a YDS schedule where all jobs have equal release times. After all, OA considers only the remaining workload and does not account for jobs not yet released. Consider such an instance, where all available jobs have release times equal to the current time t_{curr}. By definition of an interval's density, any interval of non-zero density must start at t_{curr}. From the recursive nature of YDS, we see that the speed in the resulting schedule must form a step function, as illustrated in Figure 2.1.

2.3 Survey of Relevant Speed Scaling Results

In contrast to the rather condensed and focused literature discussions in later chapters, the following section provides a more complete overview of the history and developments in energy-efficient algorithm design with respect to speed scaling. Readers questing for a more detailed overview are encouraged to take a look at the excellent survey by Albers [Alb11]. Further recommendations on this and related topics include [PST04; IP05; Alb10].

We survey the two major research directions in speed scaling: scheduling with respect to deadlines [YDS95] (cf. Chapters 3 and 4) and with respect to response time plus energy [PUW08] (cf. Chapter 5). Since the focus of Chapter 6 is not (directly) related to speed scaling, we defer the corresponding

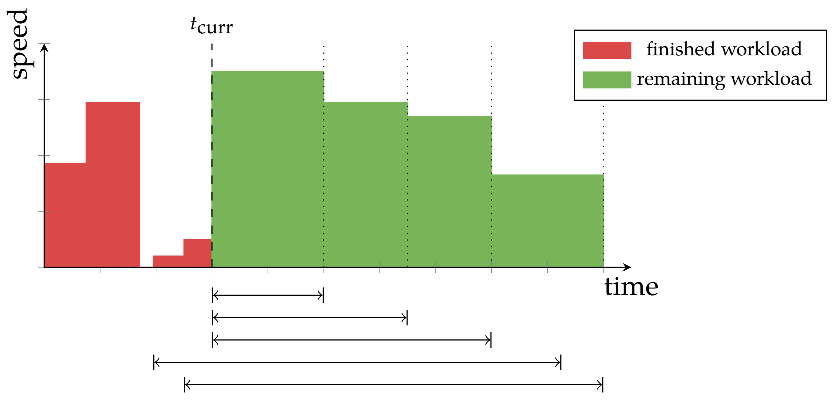


Figure 2.1: The speed (step) function (green) computed by OA at time t_{curr}. The indicated intervals correspond to the release-deadline intervals of jobs still alive and already released at time t_{curr}. For the computation of the schedule, all the release times of these jobs are effectively t_{curr}.

literature overview to that chapter.

2.3.1 Speed Scaling with respect to Deadlines

As discussed in Section 2.2, the first model and results for speed scaling are due to Yao et al. [YDS95]. Apart from the optimal offline algorithm YDS and the online algorithm OA, the authors propose and analyze another online algorithm called *Average Rate* (AVR). For every job AVR computes the *density* $\frac{p_j}{d_j-r_j}$, sets the processing speed at time t to the total density of active jobs, and uses the *Earliest Deadline First* (EDF) policy to assign jobs to the processor. Yao et al. prove that AVR's competitive ratio lies in $[\alpha^\alpha, 2^{\alpha-1}\alpha^\alpha]$. Bansal et al. [Ban+08a] tighten this by raising the lower bound to $\frac{(2-\delta)^\alpha\alpha^\alpha}{2}$, where δ depends on α and approaches zero as $\alpha \to \infty$. The same paper provides an elegant, alternative proof (based on a potential function) for AVR's competitive ratio.

Champion Algorithms Currently, the best competitive ratios for speed scaling with respect to deadlines are achieved by the algorithms BKP (for large α) and qOA (for small α). The BKP algorithm by Bansal et al. [BKP04][3] achieves a competitive ratio of $2\left(\frac{\alpha}{\alpha-1}\right)^\alpha e^\alpha$ ($\approx 2e^{\alpha+1}$ for large α). The authors also give

[3]See [BKP07] for the journal version, probably one of the most influential papers in this area.

a lower bound of $\frac{(4/3)^\alpha}{2}$ for the competitiveness of any randomized online-algorithm. The qOA algorithm is designed especially for small values of α. It is a parametrized version of OA, running at q times the speed OA would use. It is due to Bansal et al. [Ban+09] and achieves a competitive ratio of at most $\frac{4^\alpha}{2\sqrt{e\alpha}}$ (if q is set to $2 - 1/\alpha$). For $\alpha \in \{2, 3\}$, a specialized analysis yields even better results. One of the major technical contributions of this work is the introduction of a new type of potential functions, which differs considerably from the one used previously for OA and AVR.

Maximum & Discrete Speeds Chan et al. [Cha+07][4] added a new aspect to the speed scaling model by restricting the processor's maximum speed. Since such a processor may not be able to feasibly schedule all jobs, the typical objective in this case is to maximize the throughput (i.e., the total processing volume of finished jobs). Even without energy considerations, no online algorithm can be better than 4-competitive with respect to throughput [Bar+91]. Chan et al. proposed an algorithm which is $O(1)$-competitive with respect to both throughput and energy. The Slow-D algorithm by Bansal et al. [Ban+08b] achieves an optimal 4-competitiveness for throughput while maintaining a constant competitive ratio with respect to energy. In Chapter 4, we will see another example for a setting where the maximum speed is bounded.

If the processor features only a discrete set of d different speeds (similar to the model presented in Chapter 5), the YDS algorithm can be adapted to yield an optimal polynomial-time algorithm (essentially, by using the discrete speed levels to simulate an optimal continuous solution). A more efficient, direct approach by Li and Yao [LY05] computes an optimal schedule in time $O(dn \log n)$.

Impact of Sleep States Sleep states are another common energy conservation technique, capturing the phenomenon that most processors have a non-zero energy consumption when idling (e.g., to maintain the registers). To avoid this, a processor may transition into a *sleep state*, in which it has a very low (negligible) energy consumption, but resuming to work incurs additional cost (in form of time and/or energy). In the presence of sleep states, it may be

[4]See [Cha+09] for the journal version.

beneficial to run jobs at slightly higher speeds in order to enter the sleep state earlier. We will consider such sleep states in Chapter 4 of this thesis.

A theoretical model considering both sleep states and speed scaling (with arbitrary convex power functions) has first been proposed by Irani et al. [ISG03]. The authors present an offline algorithm for the problem with an approximation ratio of 3 (improved to 2 in the journal version [ISG07]) and an online algorithm based on *monotonic* online algorithms[5] for the setting without sleep states. Han et al. [Han+10] propose the online algorithm SOA (based on OA) and show that it achieves a competitive ratio of $\alpha^\alpha + 2$ if the power function has the form $P(s) = s^\alpha + \beta$. This algorithm even works well in the case of a fixed maximum speed, achieving an optimal competitive ratio for throughput and being $\alpha^\alpha + \alpha^2 4^\alpha + 2$-competitive with respect to energy. The complexity of the offline problem was recently settled by Albers and Antoniadis [AA12]. They prove NP-hardness and give improved approximation results (e.g., a $4/3$-approximation for arbitrary convex power functions).

Impact of Multiple Processors When considering multiple processors, one additional aspect to consider is whether job migration – moving a job from one to another processor – is allowed and, if so, at which cost. Without migration, a simple reduction from 3-Partition implies NP-hardness, even for identical release times and deadlines. In contrast, if migration is allowed, Chen et al. [Che+04] give an optimal polynomial-time offline algorithm.

For the non-migratory case, Albers et al. [AMS07] give a more involved NP-hardness proof, which holds even for two processors and jobs of unit size. For a variable number of processors, they prove strong NP-hardness. On the positive side, the authors propose an algorithm with an approximation ratio of $\alpha^\alpha 2^{4\alpha}$ for unit size jobs. Improved results are shown for restricted inputs which feature *agreeable deadlines*, where for any pair of jobs j, j' with $r_j \leq r_{j'}$ we must have $d_j \leq d_{j'}$. This restriction allows the formulation of an optimal polynomial-time offline algorithm for the case of unit size jobs, and to design an algorithm that computes an $\alpha^\alpha 2^{4\alpha}$-approximation in the case of non unit size jobs. Based on these offline algorithms, the authors do also provide online algorithms achieving constant competitive ratios. A result by Greiner

[5]Algorithms that increase the processor speed only at the arrival of jobs, as for example AVR.

et al. [GNS09] presents a technique yielding algorithms for multiple processors based on single processor algorithms at the cost of an additional factor of B_α in the approximation ratio and competitive ratio, respectively[6]. Similarly, for another increase of the same factor, one can transform migratory strategies to non-migratory strategies.

When migration is allowed, Bingham and Greenstreet [BG08] give an optimal polynomial-time algorithm. It is based on linear programming and, as the authors note, might have a prohibitively high complexity for most practical applications. Albers et al. [AAG11] propose a more efficient, purely combinatorial algorithm based on repeated maximum flow computations. Also, they prove that an extension of OA to multiple processors is – as in the single processor case – exactly α^α-competitive. Similarly, for an extension of the AVR algorithm, a competitive ratio of $\frac{3^\alpha\alpha^\alpha}{2} + 2^\alpha$ is proven.

Lam et al. [Lam+07] confine their view to a setting with two speed-bounded processors. They achieve a strategy which is 3-competitive concerning throughput and $O(1)$-competitive concerning energy. Note that the best possible competitive ratio for the throughput objective is 2, even if energy is of no concern.

Approaches based on Convex Programming Many results in the area of speed scaling are based on potential functions. Such arguments lead to very elegant and clean proofs once a suitable candidate for the potential has been found. However, designing such a function is often said to be more art than science, even for experienced researchers (although there are guidelines for best practices [IMP11]). In the light of this, analysis techniques based on mathematical programming and duality theory have recently gained much attention. Such techniques are not uncommon for the offline setting [BKP07; PUW08; Col+12; Ant+14], but their potential for the online setting has not been realized until about two years ago. Independently from each other, both Anand et al. [AGK12] and Gupta et al. [GKP12] used mathematical programming formulations of classical speed scaling problems together with techniques from duality theory to obtain matching competitive ratios for many problems analyzed via potential functions before. The beauty of this technique lies largely in its systematic nature. Gupta et al. [GKP12] used it to derive and analyze

[6]With B_α denoting the α-th Bell number (i.e., the number of partitions of a set of size α).

an online algorithm similar to OA that has the same competitiveness of α^α. They also prove new results for online routing in a network of speed-scalable routers. Anand et al. [AGK12] consider mainly online problems with respect to resource augmentation, improving the competitive ratios of several scheduling problems concerned with weighted response times. Following these initial results, [Ngu13] and the results presented in Chapter 3 elaborate on these techniques and extend them to other problem variants. Nguyen [Ngu13] give improved results for speed scaling on unrelated machines with respect to weighted response time plus energy (see below). In Chapter 3, we use duality theory to derive an algorithm that is α^α-competitive for the price collection version of speed scaling with deadlines on multiple processors. See that chapter for a more detailed discussion of this new approach and its potential.

2.3.2 Speed Scaling with respect to Response Time Plus Energy

Pruhs et al. [PUW04][7] initiated research on the second major research branch in energy-aware scheduling: speed scaling with respect to *flow time* (a.k.a. response time). This is a typical objective for desktop environments, where the end user expects a responsive system. There are different ways to combine the flow time objective with energy efficiency. The most common is probably to consider a linear combination of total flow time and energy. Intuitively, this allows for a user-defined trade-off between energy and responsiveness: by adjusting the energy unit in this linear combination, the user specifies how much energy she is willing to spent on decreasing the response time by one unit. We will consider such a problem in Chapter 5 of this thesis.

Running on Battery Pruhs et al. [PUW04] consider the case of unit size jobs and the objective to minimize total flow time with a fixed energy budget[8]. For this case, it is easy to see that an optimal schedule can use the *First In First Out* (FIFO) policy to assign jobs to the processor. Thus, if we order the jobs by increasing release times, we can restrict ourselves to schedules with completion times c_j such that $c_1 < c_2 < \dots < c_n$. Moreover, using the power function's convexity, one can see that each job j can be run at a

[7]See [PUW08] for the journal version.
[8]Their approach extends to the objective of flow time plus energy.

constant speed s_j (cf. [YDS95]). It is relatively easy to derive a convex program for this problem (cf. [PUW04]). However, the resulting algorithm has the drawbacks of being rather complex and is not guaranteed to run in lower order polynomial time. Thus, Pruhs et al. use the convex program only to extract structural properties of optimal schedules, which are then used to derive an efficient combinatorial algorithm. The basic idea is to classify schedules by *configurations*. A configuration $\phi\colon \mathcal{J} \to \{<,=,>\}$ is a function that specifies how the completion time c_j of each job relates to the release time r_{j+1} of its successor. For any fixed configuration ϕ, one can define a map M_ϕ mapping the available energy to the optimal total flow time for this amount of energy. Computing the lower envelope $M(A) := \min_\phi M_\phi(A)$ over all maps M_ϕ solves the scheduling problem for arbitrary energy bounds A. The basic approach to compute $M(A)$ is as follows: For large enough A, one has $c_j < r_{j+1}$ for the optimal schedule (there is enough energy to finish all jobs fast enough to avoid overlaps). Now, while continuously decreasing the available energy A, one has to find a way to (a) compute $M_\phi(A)$ for any given configuration ϕ, (b) recognize when the optimal configuration changes, and (c) find the new optimal schedule when configurations change. Details of this approach can be found in [PUW04][9]. Bunde [Bun06] extends these results to multiple processors, showing how to compute arbitrarily good approximations (still for jobs of unit size). The model and approach presented in Chapter 5 of this thesis consider a related problem for arbitrary work jobs (but with discrete speeds and a relaxed objective function). See that chapter for a more detailed and focused literature discussion on this topic.

The Online Setting Scheduling with a fixed energy budget hardly allows for good online algorithms: not having any hints about the number of future jobs, the online algorithm cannot know how much of its energy budget to invest into available jobs. Thus, Albers and Fujiwara [AF06] propose to minimize a linear combination of total flow time and energy. For the non-preemptive case and jobs of arbitrary processing volumes, they prove a lower bound of $\Omega(n^{1-1/\alpha})$ on any deterministic algorithm's competitive ratio.[10] For the case

[9]See [PUW08] for the journal version.

[10]This implies that speed scaling does not help (much) to overcome bad scheduling decisions, since this nearly matches the corresponding bound without speed scaling (which is $\Omega(n)$).

of unit size jobs, a deterministic online algorithm that achieves a constant competitive ratio of at most $8.3e(1+\Phi)^{\alpha}$ is proposed (Φ being the golden ratio). In the discussion at the end of the journal version of their paper [AF07], the authors propose a simple, natural algorithm which they believe to yield an improved performance: by setting the processor speed to $\sqrt[\alpha]{l}$, where l denotes the number of active jobs, both parts of the cost function (flow and energy) increase at the same rate l. Bansal et al. [BPS07] prove that this algorithm is in fact 4-competitive for unit size jobs. For the case of jobs with an arbitrary processing volume and total weighted flow time, the authors derive an algorithm yielding a competitive ratio of approximately $\alpha^2/\ln^2\alpha$ (for large values of α and ignoring lower order terms). It is important to note that this algorithm's speed depends on the remaining *work* of active jobs (instead of their number).

For both algorithms in [BPS07], the analysis is based on the idea of another objective, called *fractional weighted flow time plus energy*. This objective applies in situations where jobs benefit from any (infinitesimal small) completed portion of a job (instead of only fully completed jobs). See also Chapter 5, where we consider a speed scaling problem with respect to fractional weighted flow time plus energy. This objective allows for a much simpler analysis[11] and can be generalized (in the setting of [BPS07]) to the (integral) weighted flow plus energy objective at the loss of only a small factor in the competitive ratio. An interesting feature of [BPS07] is its usage of a potential function instead of directly comparing the online schedule to an optimal schedule, as was done before [PUW04; AF06].

Bounded Maximum Speed Bansal et al. [Ban+08b] transfer the aforementioned results to the setting of bounded maximum speed. More exactly, they give a 4-competitive algorithm in the case of unit size jobs and a $(2+o(1))\frac{\alpha}{\ln\alpha}$-competitive algorithm for the objective of fractional weighted flow time plus energy. If using a maximum-speed augmented processor[12], the last result can be lifted to the integral weighted flow time plus energy objective. These bounds essentially match the ones in the unbounded speed setting.

[11]For example, even for a constant speed processor, computing the optimal weighted flow schedule is NP-hard, while for fraction flow an optimal schedule is easily achieved by the *Highest Density First* (HDF) policy.

[12]Bansal and Chan [BC09] prove that there is no $O(1)$-competitive algorithm for minimizing total weighted flow time (even without energy).

Lam et al. [Lam+08] study algorithms where the speed of the online algorithm depends solely on the number of active jobs (instead of their remaining work, as is the case in [BPS07]). Such algorithms have the benefit to be easier to employ in non-clairvoyant settings (where the processing volumes of jobs are not known, even when released) and not to continuously change their speed. They propose an algorithm for the clairvoyant setting, which uses *Shortest Remaining Processing Time* (SRPT) as the job selection policy and ensures that the power matches the active number of jobs. This improves the results from [BPS07; Ban+08b] to competitive ratios of approximately $\frac{\alpha}{\ln \alpha}$ (instead of the square of this term).

Arbitrary Power Functions A result by Bansal et al. [BCP09][13] yields another generalization, considering for the first time power functions which are not of the form $\mathrm{P}(s) = s^{\alpha}$. They give a $3 + \varepsilon$-competitive algorithm[14] for total flow time plus energy and a $2 + \varepsilon$-competitive algorithm for fractional weighted flow time plus energy. Note that these are the first results for these problems with a competitive ratio that does not depend on α (even for the classical power function). The analysis is based on a new kind of potential function, inspired by the one used in [Lam+08]. Based on this, further improvements are given by Andrew et al. [AWT09], showing that the schedule using SRPT and speed $\mathrm{P}^{-1}(l)$ is 2-competitive. Furthermore, they show that this is essentially tight, since for any algorithm there is a power function yielding a matching lower bound.

[13]See [BCP13] for the journal version.
[14]SRPT for job selection and speed set to $P^{-1}(l + 1)$, where l is the number of active jobs.

CHAPTER 3

Profitable Deadline Scheduling

“*With multi-core it's like we are throwing this Hail Mary pass down the field and now we have to run down there as fast as we can to see if we can catch it.*”

David Patterson, UC Berkeley computer science professor

From an economical point of view, the value of energy has increased tremendously during the last decades. This applies not only to the energy consumed in small-scale computer systems but especially to the energy consumption in large, massively parallel data centers. According to current reports (see, for example, the report by Barroso and Hölzle [BH07]), the decisive factors regarding the costs of running a data center are mostly the energy needed for the cooling process and for the actual computations rather than the acquisition of the necessary hardware. Thus, energy efficiency has high priority for all big data center operators[1]. In order to maximize their revenue, data centers strive to minimize the energy consumption while still guaranteeing a sufficiently high quality of service to their customers. One of the most popular and widespread techniques to save energy is speed scaling, as introduced in the previous chapters. In this chapter, we study parallel

[1]Detailed information about energy-saving efforts of Apple, Facebook, and Google can be found, for example, on the corresponding websites [App13a; Fac13; Goo13].

systems (like data centers) that provide support for speed scaling. How can data centers utilize speed scaling to save energy while maintaining a sufficiently high quality of service? That is to say, can we come up with scheduling algorithms to run data centers more profitably?

Profitability Simply decreasing the speed at times of small load may lower the total energy consumption substantially. However, a lower speed often also implies a lower quality of service, which in turn may impair a data center's revenue. On a relatively basic level, we may imagine the situation for data centers as follows: Jobs of different sizes, values, and time constraints (deadlines) arrive over time at the data center. For finishing a customer's job in time, the data center receives a payment corresponding to the job's value. However, to finish a job, the data center has to invest an amount of energy depending on the job's size and time constraints. Investing into low-value jobs that require much energy lowers the profit. But even processing jobs whose values seem to justify the energy investment may be bad, as this may hinder the efficient processing of more lucrative jobs that arrive later. Thus, one has to carefully choose not only how and when to process the different jobs but also which to process at all. The remainder of this chapter studies this scenario by extending the classical speed scaling model to multiple processors and by considering an objective that incorporates profitability.

Chapter Basis The model, analysis, and results presented in the remainder of this chapter are based on the following publication:

> **2013** (with P. Pietrzyk). "Profitable Scheduling on Multiple Speed-Scalable Processors". In: *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, cf. [KP13].

Chapter Outline Before we give a formal description of the aforementioned scenario, we survey related work and describe our contribution with respect to known results in Section 3.1. Afterward, in Section 3.2, we introduce the formal model and further preliminaries related to a convex programming formulation of the presented scheduling problem. On this basis, we derive a primal-dual algorithm (PD) for the problem in Section 3.3. The main part of this chapter follows in Section 3.4, where we present and proof our main result: algorithm PD is exactly α^α-competitive (Theorem 3.3). Section 3.5 finishes this chapter with a short résumé.

3.1 Related Work & Contribution

There exists plenty of work concerning energy-efficient scheduling strategies in both theoretical and practical contexts. In the following, we concentrate on models for speed-scalable processors and jobs with deadline constraints. The literature overview presented in this section is largely self-contained and should cover the prerequisites of this chapter. If desired, the reader may consult Chapter 2 (especially Section 2.3) for a more general literature overview and some recommendations for more elaborate surveys.

Theoretical work in this area has been initiated by Yao et al. [YDS95]. They considered a single speed-scalable processor that processes preemptable jobs which arrive over time and come with different deadlines and workloads. Yao et al. studied the question of how to finish all the jobs in an energy-minimal way. In their seminal work [YDS95], they modeled the power consumption $P_\alpha(s)$ of a processor running at speed s by a constant degree polynomial $P_\alpha(s) = s^\alpha$. Here, the energy exponent α is assumed to be a constant $\alpha \geq 2$. In classical CMOS-based systems $\alpha = 3$ usually yields a suitable approximation of the actual power consumption. Yao et al. developed an optimal offline algorithm, known as YDS, as well as the two online algorithms *Optimal Available* (OA) and *Average Rate* (AVR). Up to now, OA remains one of the most important algorithms in this area, being an essential part of many algorithms for both the original problem as well as for its manifold variations. Using a rather complex but elegant amortized potential function argument, Bansal et al. [BKP04] proved that OA is exactly α^α-competitive. They also proposed a new algorithm, named BKP, which achieves a competitive ratio of essentially $2e^{\alpha+1}$. The algorithm qOA presented by Bansal et al. [Ban+09] is particularly well suited for low powers of α, where it outperforms both OA and BKP. In this work, the authors also proved that no deterministic algorithm can achieve a competitive ratio better than $e^{\alpha-1}/\alpha$. In their recent work, Albers et al. [AAG11] presented an optimal offline algorithm for the multiprocessor case. Moreover, using this algorithm, they were also able to extend OA to the multiprocessor case and proved the same competitive ratio of α^α as in the single processor case.

All results mentioned so far are only concerned with the energy necessary to finish *all* jobs. With respect to the profitability aspect, the two most relevant results for us are due to Chan et al. [CLL10a] and Pruhs and Stein [PS10]. Both proposed a model incorporating profitability into classical energy-efficient scheduling. In the simplest case, jobs have values and the scheduler is no longer required to finish all jobs. Instead, it can decide to not process jobs whose values do not justify the foreseeable energy investment necessary to complete them. The objective is to maximize the profit [PS10] or, similarly, to minimize the loss[2] [CLL10a]. As argued by the authors, the latter model has the benefit of being a direct generalization of the classical model by Yao et al. For maximizing the profit, Pruhs and Stein [PS10] showed that in order to achieve a bounded competitive ratio, resource augmentation is necessary and gave a scalable online algorithm. For minimizing the loss, Chan et al. [CLL10a] gave an $\alpha^\alpha + 2e\alpha$-competitive algorithm.

Another very important and recent work is due to Gupta et al. [GKP12] and considers the *Online Generalized Assignment Problem* (OnGAP). The authors showed an interesting relation to a multitude of problems in the context of speed-scalability (not only for scheduling). They developed a convex programming formulation of the problem and applied well-known techniques from convex optimization. In particular, they used a greedy primal-dual approach as known from linear programming (see, e.g., [BN09]). This way, they designed an online algorithm for the classical model by Yao et al. (no job values; one processor) which is very similar to OA and proved the exact same competitive ratio of α^α.

Contribution This chapter presents and studies a new online algorithm for scheduling valuable jobs on multiple speed-scalable processors. This algorithm improves upon price-collecting results from [CLL10a] in two respects:

(a) For the single processor case, it improves the best known competitive ratio from $\alpha^\alpha + 2e\alpha$ to α^α.

(b) Moreover, this constant competitive ratio holds even for the case of multiple processors.

[2]Sometimes also referred to as the *price-collecting* version of Yao et al.'s speed scaling model.

Note that this is the first algorithm that is able to handle the multiprocessor case in this scenario. We also prove that its analysis is tight.

The presented analysis is significantly different from the typical potential function argument, which is dominant in the analysis of online algorithms in this research area. It is based on a framework recently suggested by Gupta et al. [GKP12] and utilizes well-known tools from convex optimization, especially duality theory and primal-dual algorithms. We will develop a convex programming formulation and design a greedy primal-dual online algorithm. Compared to the original framework, we have to overcome the additional issue of integral variables in our convex program, which are caused by the new profitability aspect. Moreover, the handling of multiple processors proves to be a challenging task. It not only causes a much more complex objective function in the convex program but also makes it harder to grasp the structural properties of the resulting schedule. The results show that this technique is not only suitable for the classical energy-efficient scheduling model but also for more complex variants, as conjectured by Gupta et al. It is interesting to note that, in terms of the analysis, this approach goes back to the roots of Yao et al.'s model, as the optimality proof of the YDS algorithm [BKP07] is based on a similar convex programming formulation and the well-known KKT conditions from convex optimization [BV04]. Our algorithm can be seen as greedily increasing the convex program's variables while maintaining a relaxed version of these KKT conditions.

3.2 Model & Preliminaries

We consider a system of m speed-scalable processors. That is, each processor can be set to any speed $s \in \mathbb{R}_{\geq 0}$ (independently from the others). When running at speed s, the power consumption of a single processor is given by the *power function* $P_\alpha(s) = s^\alpha$. Here, the constant parameter $\alpha \in \mathbb{R}_{>1}$ is called the *energy exponent*. A problem instance consists of a set $J = \{1, 2, \dots, n\}$ of n jobs. Each job $j \in J$ is associated with a *release time* r_j, a *deadline* d_j, a *workload* w_j, and a *value* v_j. A *schedule* S describes if and how the different jobs are processed by the system. It consists of m speed functions $S_i\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ ($i \in \{1, 2, \dots, m\}$) and a job assignment policy. The speed function S_i dictates the speed $S_i(t)$ of the i-th processor at time t. The job assignment policy decides which jobs to run on the processors. At any time t, it may schedule at most one job per processor, and each job can be processed by at most one processor at any given time (i.e., we consider nonparallel jobs). Moreover, jobs are preemptive: a running job may be interrupted at any time and continued later on, possibly on a different processor. The total work processed by processor i between time t_1 and t_2 is $\int_{t_1}^{t_2} S_i(t)\,\mathrm{d}t$. Similarly, the overall power consumed by this processor during the same time is $\int_{t_1}^{t_2} \mathrm{P}_\alpha\,(S_i(t))\,\mathrm{d}t$. Let $s_j(t)$ denote the speed used to process job j at time t. We say job j is *finished under schedule S* if S processes (at least) w_j units of j's work during the interval $[r_j, d_j)$. That is, if we have $\int_{r_j}^{d_j} s_j(t)\,\mathrm{d}t \geq w_j$.

A given schedule S may not finish all n jobs. In this case, the total value of unfinished jobs is considered as a loss. Thus, the cost of S is defined as the sum of the total energy consumption and the total value of unfinished jobs. More formally, if J_{rej} denotes the set of unfinished (rejected) jobs under schedule S, we define the *cost of schedule S* by

$$\mathrm{cost}(S) := \sum_{i=1}^{m} \int_0^\infty \mathrm{P}_\alpha\,(S_i(t))\,\mathrm{d}t + \sum_{j\in J_{\mathrm{rej}}} v_j. \tag{3.1}$$

Our goal is to construct a low-cost schedule in the *online scenario* of the problem. That is, the job set J is not known a priori, but rather revealed over time. In particular, we do not know the total number of jobs, and the existence as well as the attributes of a job $j \in J$ are revealed just when the job is released at time r_j. We measure the quality of algorithms for this online problem by their *competitive ratio*: Given an online algorithm A, let $A(J)$ denote the resulting schedule for job set J. The competitive ratio of A is defined as

$$\sup_J \frac{\mathrm{cost}(A(J))}{\mathrm{cost}(\mathrm{OPT}(J))}, \tag{3.2}$$

where $\mathrm{OPT}(J)$ denotes an optimal schedule for the job set J. Note that, by definition, the competitive ratio is at least one.

3.2.1 Convex Programming Formulation

In the following, we develop a convex programming formulation of the above (offline) scheduling problem to aid us in the design and analysis of our on-

$$\begin{array}{llr}
\min\limits_{\substack{0 \leq x \\ y \in \{0,1\}^n}} & \sum\limits_{k=1}^{N} \mathcal{P}_k(x_{1k}, x_{2k}, \dots, x_{nk}) + \sum\limits_{j \in J} (1 - y_j) v_j & \\
\text{s.t.} & y_j - \sum\limits_{k=1}^{N} c_{jk} x_{jk} \leq 0 & \forall j \in J
\end{array}$$

Figure 3.1: Mathematical programming formulation (IMP) for profitable deadline scheduling on multiple processors.

line algorithm (cf. Section 3.3). Following an idea by Bingham and Greenstreet [BG08], we partition time into *atomic intervals* T_k using the jobs' release times and deadlines. The goal of our convex program is to compute what portion of each job to process during the different atomic intervals in an optimal schedule. Once we have such a fixed *work assignment*, we use a deterministic algorithm by Chen et al. [Che+04] to efficiently compute an energy-minimal way to process the corresponding work on the m processors in this interval. The energy consumption of the resulting schedule in the interval T_k can be written as a convex function $\mathcal{P}_k$ of the work assignment. This function plays a crucial role in the optimization objective of our convex program, and studying its properties and the corresponding schedule's structure is an important part of our analysis. We will elaborate on $\mathcal{P}_k$ once we have derived the convex program (see Section 3.2.2).

For a given job set J, let us partition the time horizon into $N \in \mathbb{N}$ atomic intervals T_k ($k \in \{1, 2, \dots, N\}$) as follows. We define $T_k := [\tau_{k-1}, \tau_k)$ where $\tau_0 < \tau_1 < \cdots < \tau_N$ are chosen such that $\{\tau_0, \tau_1, \dots, \tau_N\} = \{r_j, d_j \mid j \in J\}$. Let $l_k := \tau_k - \tau_{k-1}$ denote the length of interval T_k. Note that there are at most $2n - 1$ intervals. To model the deadline constraint of job j, we introduce parameters $c_{jk} \in \{0, 1\}$ that indicate whether $T_k \subseteq [r_j, d_j)$ ($c_{jk} = 1$) or not ($c_{jk} = 0$). Our program uses two types of variables: *load variables* $x_{jk} \in [0, 1]$ for each job $j \in J$ and each atomic interval $k \in \{1, 2, \dots, N\}$, and *indicator variables* $y_j \in \{0, 1\}$ for each job $j \in J$. The variable x_{jk} indicates what portion of j's workload is assigned to interval T_k and the variable y_j indicates whether job j is finished ($y_j = 1$) or not ($y_j = 0$). Figure 3.1 shows the complete (integral) mathematical program (IMP) for our scheduling problem. The first summand in the objective corresponds to the energy spent in the different intervals. The second summand charges costs for all unfinished jobs. The set of constraints ensures that a job can be declared as finished only if it has been completely assigned to intervals T_k lying in its release-deadline interval $[r_j, d_j)$. We use x and y to refer to the full vectors of variables x_{jk} and y_j, and we use the symbol "$\preceq$" for element-wise comparison.

If we relax the domain of (IMP) such that $0 \preceq y \preceq 1$, we get a convex program. We refer to this convex program as (CP). By introducing dual variables λ_j (also called *Lagrange multipliers*) for each constraint of (CP) we can write its *Lagrangian* $L(x, y, \lambda)$ as

$$\sum_{k=1}^{N} \mathcal{P}_k(x_{1k}, x_{2k}, \dots, x_{nk}) + \sum_{j \in J} (1 - y_j) v_j + \sum_{j \in J} \lambda_j \left(y_j - \sum_{k=1}^{N} c_{jk} x_{jk} \right). \tag{3.3}$$

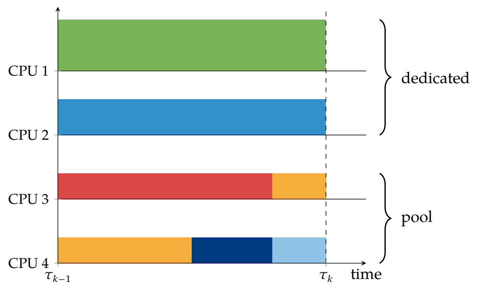
It is a linear combination of the convex program's objective and constraints. Instead of prohibiting infeasible solutions (as done by the convex program), it charges a penalty for violated constraints (assuming positive λ_j). Now, the *dual function* of (CP) is defined as

$$g(\lambda) := \inf_{\substack{0 \preceq x \\ 0 \preceq y \preceq 1}} L(x, y, \lambda). \tag{3.4}$$

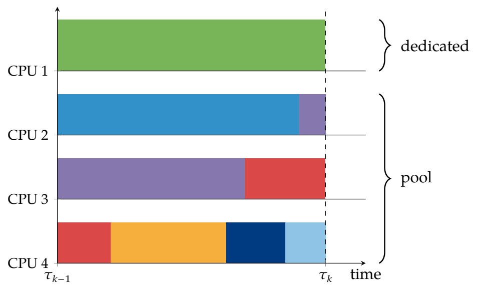
An important property of the dual function g is that for any $\lambda \succeq 0$, the value $g(\lambda)$ is a lower bound on the optimal value of (CP). Moreover, since (CP) is a relaxation of (IMP), $g(\lambda)$ is also a lower bound on the optimal value of (IMP). See the book by Boyd and Vandenberghe [BV04] for further details on these and similar known facts about (convex) optimization problems.

3.2.2 Power Consumption in Atomic Intervals

Let us give a more detailed description of the function $\mathcal{P}_k(x_{1k}, x_{2k}, \dots, x_{nk})$. We defined $\mathcal{P}_k$ implicitly by mapping a given work assignment $x_{1k}, x_{2k}, \dots, x_{nk}$ for interval T_k to the power consumption of Chen et al.'s algorithm [Che+04] during T_k. This guarantees an energy-minimal schedule for the given work assignment. In the following, we give a concise description of this algorithm and derive a more explicit formulation as well as some properties of $\mathcal{P}_k$.



(a) Before the arrival of a new job.



(b) After the arrival of a new job.

Figure 3.2: Schedules computed by Chen et al.'s algorithm before and after the arrival of a new job.

To ease the discussion, let us assume that the jobs are numbered such that $x_{1k}w_1 \geq x_{2k}w_2 \geq \dots \geq x_{nk}w_n$. In a nutshell, Chen et al.'s algorithm can be described as follows. Define the job set

$$\psi(k) := \left\{ j \in J \;\middle|\; j \leq m \;\wedge\; x_{jk} > 0 \;\wedge\; x_{jk}w_j \geq \frac{\sum_{j'>j} x_{j'k}w_{j'}}{m-j} \right\}. \tag{3.5}$$

These jobs are called *dedicated jobs* and are scheduled on their own *dedicated processor* using the energy-optimal (since minimal) speed $s_{jk} := \frac{x_{jk}w_j}{l_k}$. All remaining jobs, called *pool jobs*, are scheduled on the remaining *(pool) processors* in a greedy manner. The intuition is that dedicated jobs are larger than the remaining average workload and thus must be processed on a dedicated processor. See [BG08, Section 3.1] for a relatively short but more detailed description of the algorithm. Figure 3.2 illustrates the resulting schedule and how it may change due to the arrival of a new job. Using the above definition of dedicated jobs we can write $\mathcal{P}_k$ as

$$\mathcal{P}_k(x_{1k}, \dots, x_{nk}) = \sum_{j \in \psi(k)} l_k \, \mathrm{P}_\alpha\left(\frac{x_{jk}w_j}{l_k}\right) + (m - |\psi(k)|) l_k \, \mathrm{P}_\alpha\left(\frac{\sum_{j \notin \psi(k)} x_{jk}w_j}{(m - |\psi(k)|) l_k}\right). \tag{3.6}$$

The following proposition gathers some important properties concerning the power consumption function $\mathcal{P}_k$ of an atomic interval T_k.

Proposition 3.1. *Consider an arbitrary atomic interval T_k together with its power consumption function $\mathcal{P}_k \colon \mathbb{R}^n_{\geq 0} \to \mathbb{R}$. This function has the following properties:*

(a) *It is convex and* $\mathcal{P}_k(0) = 0$.

(b) *It is differentiable with partial derivatives* $\frac{\mathrm{d}\,\mathcal{P}_k}{\mathrm{d}x_{jk}}(x_{1k}, \dots, x_{nk}) = w_j \cdot \mathrm{P}'_\alpha\big(s_{jk}\big)$. *Here,* s_{jk} *denotes the speed used to schedule the workload* $x_{jk}w_j$ *in Chen et al.'s algorithm:*

$$s_{jk} = \begin{cases} {}^{x_{jk}w_j}/_{l_k} & \text{, if } j \text{ is a dedicated job} \\ \frac{\sum_{j \notin \psi(k)} x_{jk}w_j}{(m - |\psi(k)|)l_k} & \text{, if } j \text{ is a pool job.} \end{cases} \tag{3.7}$$

Proof Sketch.

(a) The equality $\mathcal{P}_k(0) = 0$ is obvious from the definition of $\mathcal{P}_k$. The convexity follows from [BG08, Lemma 3.2]. There, the authors proved the convexity of $(x_{1k}, \dots, x_{nk}) \mapsto \mathcal{P}_k(x_{1k}/w_1, \dots, x_{nk}/w_n)$ (a linear transformation of $\mathcal{P}_k$).

(b) Differentiability is obvious for all points $(x_{1k}, \dots, x_{nk})$ for which all the inequalities $x_{jk}w_j \geq {}^{\sum_{j' \geq j} x_{j'k}w_{j'}}/_{(m-j)}$ in Equation (3.5) are strict: For these, we have a small interval around x_{jk} such that the set $\psi(k)$ of dedicated jobs does not change. Differentiability follows by noting that $\mathcal{P}_k$ is given by Equation (3.6) on these intervals.

For other points, one can compute the left and right derivatives in x_{jk}, distinguishing whether job j switched between a dedicated processor and a pool processor, whether j stays on a dedicated processor, or whether j stays on a pool processor and some other jobs switch between processor types. All cases yield the same left and right derivatives as given in the statement. □

We will also need to compare the result of Chen et al.'s algorithm before and after the arrival of a new job (cf. Figure 3.2). That is, how can the workloads on the processors change when a single entry of the work assignment changes from zero to some positive value?

Proposition 3.2. *Consider Chen et al.'s algorithm executed twice for some interval* T_k *with the work assignments* $x = (x_1, x_2, \dots, x_n, 0)$ *and* $x' = (x_1, x_2, \dots, x_n, z)$, *respectively (i.e., before and after the arrival of a new job). Let* L_i *and* L'_i *denote the total workload on the i-th fastest processor in the resulting schedules, respectively. Then we have* $0 \leq L'_i - L_i \leq z$.

Proof Sketch. We consider only the normalized case. That is, the case of unit workloads ($w_j = 1$ for all jobs) and an atomic interval of unit length ($l_k = 1$). The general case follows by a straightforward adaption. Without loss of generality, we furthermore assume $x_1 \geq x_2 \geq \dots \geq x_n$. Note that we do not presume any relation between the newly arrived workload z and the remaining workloads. Let S and S' be the schedules produced by Chen et al.'s algorithm for the work assignments x and x', respectively. Similarly, we use d and d' to denote the number of dedicated processors, and L_{pool} and L'_{pool} for the workload of a pool processor in S and S', respectively. Remember that pool processors have the smallest workload. That is, we have $L_i \geq L_{\text{pool}}$ and $L'_i \geq L'_{\text{pool}}$ for all $i \in \{1, 2, \dots, m\}$.

We start with the proof of $L'_i - L_i \geq 0$. Observe that the arrival of the workload z will not cause any of the former pool jobs to become a dedicated job (cf. Equation (3.5)). Moreover, by the same equation, for each dedicated processor that becomes a pool processor we also get a new pool job that has a workload of at least L_{pool}. Thus, the workload of pool processors from S can only increase. The workload of the i-th fastest dedicated processor in S is exactly x_i. If it becomes a pool processor, we have $x_i < L'_{\text{pool}} = L'_i$, yielding $L_i = x_i < L'_i$. If it stays a dedicated processor, its workload is the i-th largest value in $\{x_1, \dots, x_n, z\}$ and, thus, at least as large as the i-th largest value in $\{x_1, \dots, x_n\}$, yielding $L_i \leq L'_i$. To prove the second statement, $L'_i - L_i \leq z$, let us assume $L'_i - L_i > z$ and seek a contradiction. We distinguish two cases, depending on the type (pool or dedicated) of the i-th fastest processor in S':

Processor i is a pool processor in S': Note that $z < L'_i - L_i \leq L'_i$ and i being a pool processor implies that z is also scheduled on a pool processor (cf. Equation (3.5)). As d' is the number of dedicated processors, we must have $i > d'$. Moreover, all the jobs with workload less than $L'_{d'}$ must be pool jobs in S'. These are exactly the jobs which are scheduled on the processors $d' + 1, \dots, m$ in schedule S. Thus, the total workload of all pool processors in S' equals $(m - d')L'_i = z + \sum_{j>d'} L_j$. Using $i > d'$, $L'_{i'} - L_{i'} \geq 0$ for all $i' \in \{1, 2, \dots, m\}$, and that all pool processors in S' have the same workload, we get $z = (m - d')L'_i - \sum_{j>d'} L_j = \sum_{j>d'}(L'_i - L_j) = \sum_{j>d'}(L'_j - L_j) \geq L'_i - L_i$. This contradicts our assumption.

Processor i is a dedicated processor in S': Our assumption implies $L'_i > L_i + z \geq z$. Together with i being a dedicated processor this yields $L'_i = x_i$ (because x_i remains the i-th largest value in $\{x_1, x_2, \dots, x_n, z\}$). But the assumption also implies $L'_i > L_i + z \geq L_i \geq x_i$. We get the contradiction $x_i = L'_i > x_i$. □

3.3 An Online Greedy Primal-Dual Algorithm

The goal of this section is to use the convex programming formulation (CP) and its dual function $g\colon \mathbb{R}^n \to \mathbb{R}$ to derive a provably good online algorithm for our scheduling problem. We start by describing an algorithm that computes a solution to (CP) in an online fashion, but knowing the time partitioning T_k ($k \in \{1, 2, \dots, n\}$). Subsequently, we explain how this solution is used to compute the actual schedule and how we handle the fact that the actual atomic intervals are not known beforehand. To solve (CP), we use a greedy primal-dual approach for convex programs as suggested by Gupta et al. [GKP12]. Our algorithm extends their framework to the multiprocessor case and to profitable scheduling models. It shows how to incorporate rejection policies into the framework (handling the integral constraints in the convex program) and how to cope with more complex power functions of a system (in our case $\mathcal{P}_k$).

The Primal-Dual Algorithm Our primal-dual algorithm, in the following referred to as PD, maintains a set of primal variables (x, y) and a set of dual variables λ, all initialized with zero. Whenever a new job (i.e., a constraint of (CP)) arrives, we start to increase the primal variables x_{jk} ($k \in \{1, 2, \dots, N\}$) in a greedy fashion until either the full job is scheduled (i.e., $\sum_k x_{jk} = 1$) or the planned energy investment for job j becomes too large compared to its value. In the latter case, the variables x_{jk} are reset to zero, λ_j is set to v_j, and y_j remains zero (the job is rejected). Otherwise, we set y_j to one (the job is finished) and λ_j to essentially the current rate of cost increase per job workload. When greedily increasing the primal variables, we assign the next infinitesimal small portion of job j to those atomic intervals that cause the smallest increase in costs. Essentially, these are the intervals where j's workload would be scheduled with the slowest speed. See Listing 3.1 for the algorithm.

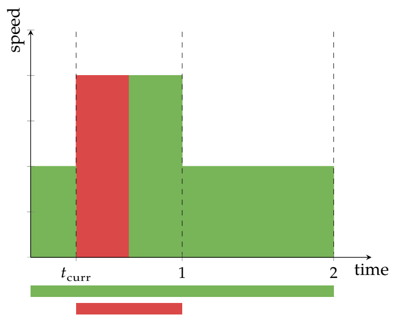
The described algorithm is similar to primal-dual algorithms known from

```
1  {executed each time a new job $j \in J$ arrives}
2  initialize $x_{jk}, y_j$, and $\lambda_j$ with zero for all $k \in \{1, 2, \dots, N\}$
3  for each interval $T_k \subseteq [r_j, d_j)$ compute
4      $\lambda_{jk} := \delta \frac{\mathrm{d}\,\mathcal{P}_k}{\mathrm{d}x_{jk}}(x_{1k}, x_{2k}, \dots, x_{jk}, 0, \dots, 0)$
5
6  let the set $\mathcal{T}_{\min}$ contain all $T_k$ with minimal $\lambda_{jk}$
7  for each $T_k \in \mathcal{T}_{\min}$ in parallel:
8      increase $x_{jk}$ in a continuous way (which in turn raises $\lambda_{jk}$ according to line 3)
9      ensure that all $\lambda_{jk}$ of intervals in $\mathcal{T}_{\min}$ remain equal
10     update $\mathcal{T}_{\min}$ whenever the $\lambda_{jk}$ reach a $\lambda_{jk'}$ with $T_{k'} \notin \mathcal{T}_{\min}$
11     stop increasing once one of the following comes true
12     (a) $\sum x_{jk} = 1$ :      set    $y_j := 1, \lambda_j := \lambda_{jk}$
13     (b) $\lambda_{jk} = v_j$ :   reset  $x_{jk} := 0, \lambda_j := \lambda_{jk}$
```

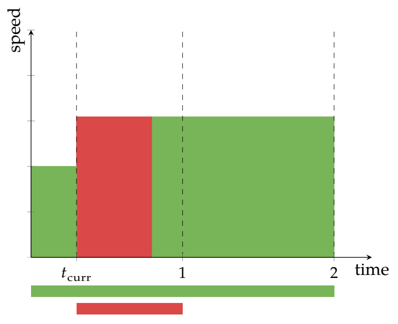
Listing 3.1: Primal-dual algorithm PS with parameter δ.

linear programming, where primal and/or dual variables are raised at certain rates until the (relaxed) complementary slackness conditions are met. In fact, this algorithm is derived by using relaxed versions of the Karush-Kuhn-Tucker (KKT) conditions, essentially a generalization of the complementary slackness conditions for convex (or even general nonlinear) programs. The actual schedule used is the one computed by Chen et al.'s algorithm when applied to the current work assignment given by the primal variables x_{jk} for the atomic interval T_k.

Concerning the Time Partitioning Our algorithm formulation assumes a priori knowledge of the atomic intervals T_k. However, since the jobs arrive in an online fashion, the exact partitioning is actually not known to the algorithm. One can reformulate the algorithm such that it uses the intervals T'_k induced by the jobs $J' = \{1, 2, \dots, j\} \subseteq J$ it knows so far. If a refinement of an atomic interval $T'_k = T_{k_1} \cup T_{k_2}$ occurs due to the arrival of a new job, the already assigned job portions are simply split according to the ratios $|T_{k_1}|/|T'_k|$ and $|T_{k_2}|/|T'_k|$. This reformulated algorithm produces an identical schedule. To see this, note that the algorithm with a priori knowledge of the refinement $T'_k = T_{k_1} \cup T_{k_2}$ treats both intervals T_{k_1} and T_{k_2} as identical (with respect to their relative size $|T_{k_i}|/|T'_k|$) up to the point when the job causing the refinement arrives.



(a) PD Schedule



(b) OA Schedule

Figure 3.3: The dashed lines indicate atomic intervals, the horizontal bars the jobs' availability. Note that PD's schedule is more conservative in comparison, leaving more room for scheduling jobs that might occur during the last atomic interval.

Relation to the OA Algorithm For the case of a single processor and sufficiently high job values, algorithm PD is quite similar to the popular OA algorithm by Yao et al. [YDS95]. When a new job arrives, PD essentially finds the atomic intervals of lowest speed and increases their speed to free computational resources to be used for the new job. This is also true for the OA algorithm. However, while PD never changes how other jobs are distributed over atomic intervals, OA may actually influence this distribution. Figure 3.3 gives a simple example for the structural difference of the resulting schedules. Another interesting observation is that, in the single processor case, our analysis yields the very same optimal rejection policy as an OA-based algorithm by Chan et al. [CLL10a]. Indeed, as we will see in Section 3.4, our analysis yields that $\delta = \alpha^{1-\alpha}$ is the optimal choice for the parameter δ. Using this parameter, one can easily check that our rejection policy essentially states to reject a job if its energy consumption in the planned schedule exceeds $\alpha^{\alpha-2} \cdot v_j$. Or, equivalently, a job is rejected if its speed in the planned schedule exceeds $\alpha^{\frac{\alpha-2}{\alpha-1}} \cdot (v/w)^{\alpha-1}$, the rejection policy from [CLL10a].

3.4 Analysis

In the following, let $(\tilde{x}, \tilde{y})$ and $\tilde{\lambda}$ denote the primal and dual variables computed by our algorithm PD. Remember that the final schedule computed by PD is derived by applying Chen et al.'s algorithm to the $\tilde{x}_{1k}, \dots, \tilde{x}_{nk}$ values in each atomic interval T_k. We refer to this schedule as the $(\tilde{x}, \tilde{y})$-schedule or simply as the schedule PD. Our goal is to use $g(\tilde{\lambda})$ to bound the cost of this schedule (referred to as $\mathrm{cost}(\mathrm{PD})$). Our main result is

Theorem 3.3. *The competitive ratio of algorithm PD with the parameter δ set to $1/\alpha^{\alpha-1}$ is at most α^α. Moreover, there is a problem instance for which PD is exactly by a factor of α^α worse than an optimal algorithm. That is, our upper bound is optimal.*

For the upper bound, we show that $\mathrm{cost}(\mathrm{PD}) \le \alpha^\alpha g(\tilde{\lambda})$. Since, by duality, $g(\tilde{\lambda})$ is also a lower bound on the optimal value of (CP) and, thereby, on the optimal value of (IMP), we get $\frac{\mathrm{cost}(\mathrm{PD})}{\mathrm{cost}(\mathrm{OPT})} \le \alpha^\alpha$. The lower bound follows from a known result for traditional energy-efficient scheduling (without job values but the necessity to finish all jobs) by setting the job values sufficiently high.

In the remainder, we develop the key ingredients for the proof of Theorem 3.3. We start in Section 3.4.1 and derive a more explicit formulation of the dual function value $g(\tilde{\lambda})$ by relating it to a certain (infeasible) solution to our convex program (CP) and a corresponding schedule. Section 3.4.2 further simplifies this formulation by expressing $g(\tilde{\lambda})$ solely in terms of the jobs (instead of their workloads in different atomic intervals). Based on this job-centric formulation, Section 3.4.3 develops different bounds for the dual function value depending on certain job characteristics. The actual proof of Theorem 3.3 combines these bounds and can be found in Section 3.4.4.

3.4.1 Structure of an Optimal Infeasible Solution

First of all, note that the value $g(\tilde{\lambda}) = \inf L(x, y, \tilde{\lambda})$ (cf. Equation (3.4)) is finite and obtained by a pair $(\hat{x}, \hat{y})$ of primal variables. These primal variables can be interpreted as a (possibly infeasible) solution to the convex program (CP). Moreover, for our fixed dual variable $\tilde{\lambda}$, this solution is optimal in that it minimizes the sum of the objective cost and the penalty for violated constraints. In this sense, we refer to $(\hat{x}, \hat{y})$ as an *optimal infeasible solution*. Our goal is to understand the structure of this solution, which will eventually allow us to write $g(\tilde{\lambda})$ in a more explicit way. The results of this subsection are related to results from [GKP12], but more involved due to the more complex nature of our objective function.

Note that $\hat{x}$ and $\hat{y}$ may differ largely from $\tilde{x}$ and $\tilde{y}$. However, the following lemmas show a strong correlation between this optimal infeasible solution and the feasible (partially integral) solution computed by algorithm PD.

Lemma 3.4. *Consider an optimal infeasible solution* $(\hat{x}, \hat{y})$. *Without loss of generality, we can assume that it has the following properties:*

(a) The equality $\hat{y} = \tilde{y}$ *holds.*

(b) For any atomic interval T_k, *there are at most* m *different jobs* j *with* $\hat{x}_{jk} > 0$.

Proof.

(a) Consider an arbitrary job $j \in J$ and remember that the domain for the variables $\hat{y}_j$ is restricted to $[0, 1]$. The contribution of variable $\hat{y}_j$ to $g(\tilde{\lambda}) = L(\hat{x}, \hat{y}, \tilde{\lambda})$ is exactly $\hat{y}_j(\tilde{\lambda}_j - v_j)$, as can be seen by considering Equation (3.3). If $\tilde{\lambda}_j < v_j$, this is minimized by choosing $\hat{y}_j$ maximal ($\hat{y}_j = 1$). Otherwise, we must have $\tilde{\lambda}_j = v_j$ (by the definition of algorithm PD). This allows us to choose $\hat{y}_j$ arbitrarily, such that we can set it to zero. Both choices correspond exactly to the way $\tilde{y}_j$ is set by algorithm PD.

(b) Assume there are more than m jobs with $\hat{x}_{jk} > 0$. We can assume $c_{jk} = 1$ for these jobs, because otherwise we could set $\hat{x}_{jk} = 0$ without increasing $g(\tilde{\lambda}) = L(\hat{x}, \hat{y}, \tilde{\lambda})$. Now, the values $\hat{x}_{1k}, \dots, \hat{x}_{nk}$ correspond to a work assignment for the atomic interval T_k, as used by Chen et al.'s algorithm (cf. Section 3.2.2). By Equation (3.3), the contribution of these values to $g(\tilde{\lambda}) = L(\hat{x}, \hat{y}, \tilde{\lambda})$ is given by $\mathcal{P}_k(\hat{x}_{1k}, \dots, \hat{x}_{nk}) - \sum_{j \in J} \tilde{\lambda}_j \hat{x}_{jk}$. Since there are more than m jobs j with nonzero $\hat{x}_{jk}$, at least two of them must share a processor in the schedule computed by Chen et al.'s algorithm for this work assignment. In other words, there are two pool jobs $j, j' \in J \setminus \psi(k)$ with $\hat{x}_{jk}, \hat{x}_{j'k} > 0$. Together with Equation (3.6), we see that the contribution of $\hat{x}_{jk}$ and $\hat{x}_{j'k}$ to $g(\tilde{\lambda})$ consists of two terms: a convex term

$$(m - |\psi(k)|) l_k \, \mathrm{P}_\alpha \left(\frac{\sum_{j \notin \psi(k)} \hat{x}_{jk} w_j}{(m - |\psi(k)|) l_k} \right) \tag{3.8}$$

and a linear term $-\tilde{\lambda}_j \hat{x}_{jk} - \tilde{\lambda}_{j'} \hat{x}_{j'k}$. By changing $\hat{x}_{jk}$ and $\hat{x}_{j'k}$ along the line that keeps the sum $\hat{x}_{jk} w_j + \hat{x}_{j'k} w_{j'}$ constant, we can decrease one of the variables (say $\hat{x}_{jk}$) and increase the other such that the first (convex) term remains constant and the second (linear) term is not increased. This will not effect the type (dedicated or pool) of other jobs. The only job that may change its type is job j', as it may become a dedicated job. Once this happens, we iterate the process with two other pool jobs. As the number of dedicated jobs is upper bounded by m, this can happen only finitely often. Thus, at some point we can decrease $\hat{x}_{jk}$ all the way to zero without increasing the dual function value $g(\tilde{\lambda})$. We continue eliminating $\hat{x}_{jk}$ variables until at most m of them are nonzero. □

Given an atomic interval T_k, we call the jobs j with $\hat{x}_{jk} > 0$ the *contributing jobs of* T_k and denote the corresponding job set by $\varphi(k)$. As done in the proof of Lemma 3.4, we can consider $\hat{x}$ as a work assignment for the atomic intervals T_k. By applying Chen et al.'s algorithm, we get a schedule whose energy cost in interval T_k is exactly $\mathcal{P}_k(\hat{x}_{1k}, \dots, \hat{x}_{nk})$. We refer to this schedule as the $(\hat{x}, \hat{y})$-schedule. Using this terminology, the second statement of Lemma 3.4 essentially says that in this schedule at most m jobs are scheduled in any atomic interval T_k. Moreover, it follows immediately from the description of Chen et al.'s algorithm that all contributing jobs are dedicated jobs of the corresponding atomic interval.

We can derive a slightly more explicit characterization of the contributing jobs $\varphi(k)$ of an atomic interval T_k by exploiting that $(\hat{x}, \hat{y})$ is a minimizer of $(x, y) \mapsto L(x, y, \tilde{\lambda})$.

Lemma 3.5. *Consider any atomic interval T_k and its contributing jobs $\varphi(k)$. Define the value $\hat{s}_j := (\tilde{\lambda}_j/\alpha w_j)^{\frac{1}{\alpha-1}}$ for any job j.*

(a) For any $j \in \varphi(k)$ we have

$$\hat{x}_{jk} = \frac{l_k}{w_j}\hat{s}_j = \frac{l_k}{w_j}(\tilde{\lambda}_j/\alpha w_j)^{\frac{1}{\alpha-1}}. \tag{3.9}$$

Moreover, j is scheduled at constant speed $\hat{s}_j$ in the $(\hat{x}, \hat{y})$-schedule.

(b) The total contribution of the $\hat{x}_{jk}$ variables to the dual function value $g(\tilde{\lambda})$ is

$$(1-\alpha)l_k \sum_{j \in \varphi(k)} \left(\frac{\hat{x}_{jk} w_j}{l_k}\right)^{\alpha} = (1-\alpha)l_k \sum_{j \in \varphi(k)} \hat{s}_j^{\alpha}. \tag{3.10}$$

(c) *Let n_k denote the number of jobs available in the atomic interval T_k (i.e., jobs with $c_{jk} = 1$). The contributing jobs $\varphi(k)$ are the* $\min(m, n_k)$ *jobs with maximal $\hat{s}_j$-values under all available jobs.*

Proof.

(a) By definition, $\hat{x}$ is a minimizer of $x \mapsto L(x, \hat{y}, \tilde{\lambda})$. This implies that we must have $\frac{\mathrm{d}L}{\mathrm{d}x_{jk}}(\tilde{\lambda}, \hat{x}, \hat{y}) = 0$ for any contributing job $j \in \varphi(k)$. We get

$$\begin{aligned}0 = \frac{\mathrm{d}L}{\mathrm{d}x_{jk}}(\tilde{\lambda}, \hat{x}, \hat{y}) &= \frac{\mathrm{d}\,\mathcal{P}_k}{\mathrm{d}x_{jk}}(\hat{x}_{1k}, \dots, \hat{x}_{nk}) - \tilde{\lambda}_j \\ &= w_j \cdot \mathrm{P}'_\alpha\left(\frac{\hat{x}_{jk}w_j}{l_k}\right) - \tilde{\lambda}_j = \alpha w_j \left(\frac{\hat{x}_{jk}w_j}{l_k}\right)^{\alpha-1} - \tilde{\lambda}_j,\end{aligned} \tag{3.11}$$

which yields the first statement by rearranging. The second statement follows from this by noticing that $\frac{\hat{x}_{jk}w_j}{l_k}$ is the speed used by Chen et al.'s algorithm for the (dedicated) job j.

(b) By definition of $g(\tilde{\lambda}) = L(\hat{x}, \hat{y}, \tilde{\lambda})$, we get that the total contribution of the $\hat{x}_{jk}$ variables is (there are no pool jobs!)

$$\begin{aligned}\mathcal{P}_k(\hat{x}_{1k}, \dots, \hat{x}_{nk}) - \sum_{j \in \varphi(k)} \tilde{\lambda}_j \hat{x}_{jk} &= \sum_{j \in \varphi(k)} l_k \,\mathrm{P}_\alpha\left(\frac{\hat{x}_{jk}w_j}{l_k}\right) - \sum_{j \in \varphi(k)} \tilde{\lambda}_j \hat{x}_{jk} \\ = l_k \sum_{j \in \varphi(k)} \mathrm{P}_\alpha\left(\hat{s}_j\right) - \alpha l_k \sum_{j \in \varphi(k)} \frac{\tilde{\lambda}_j}{\alpha w_j} \hat{s}_j &= (1-\alpha) l_k \sum_{j \in \varphi(k)} \hat{s}_j^\alpha.\end{aligned} \tag{3.12}$$

(c) The contributing jobs must be chosen such that their contribution is minimized. Using statement (b) and $\alpha > 1$, we see that this is the case when choosing the maximal number of available jobs (at most m) with the largest $\hat{s}_j$-values. □

3.4.2 A Job-centric Formulation of the Dual Function

In the following, we assume that the optimal infeasible solution $(\hat{x}, \hat{y})$ adheres to Lemma 3.4. That is, we have $\hat{y} = \tilde{y}$ and we can relate the optimal infeasible solution to the $(\hat{x}, \hat{y})$-schedule which schedules in each atomic interval T_k exactly the $|\varphi(k)| (\leq m)$ available jobs with the largest $\hat{s}_j = (\tilde{\lambda}_j/\alpha w_j)^{\frac{1}{\alpha-1}}$-values, each on its own dedicated processor at speed $\hat{s}_j$. We use the somewhat lax notation $k \in \varphi^{-1}(j)$ to refer to the atomic intervals T_k to which j contributes. Our main goal in this section is to derive a formulation of the dual function value solely in terms of the jobs. We will also define and discuss the *trace* of a job, which helps to relate any job (even if unfinished) to a certain amount of energy consumed by our PD algorithm.

Given a job $j \in J$, let $l(j) := \sum_{k \in \varphi^{-1}(j)} l_k$ denote the total time it is scheduled in the $(\hat{x}, \hat{y})$-schedule. Moreover, let $E_{\tilde{\lambda}}(j)$ denote the total energy invested by the $(\hat{x}, \hat{y})$-schedule into job j. Now, we can formulate the following lemma.

Lemma 3.6. *For any job $j \in J$, the total energy invested by the optimal infeasible solution into job j is $E_{\tilde{\lambda}}(j) = l(j)\hat{s}_j^{\alpha}$. Moreover, the dual function value $g(\tilde{\lambda})$ can be written as*

$$g(\tilde{\lambda}) = (1-\alpha)\sum_{j \in J} E_{\tilde{\lambda}}(j) + \sum_{j \in J} \tilde{\lambda}_j. \tag{3.13}$$

Proof. The equality $E_{\tilde{\lambda}}(j) = l(j)\hat{s}_j^{\alpha}$ follows immediately from the above definitions, as j is processed by the $(\hat{x}, \hat{y})$-schedule at constant speed $\hat{s}_j$ for a total time of exactly $l(j)$. For the lemma's main statement, remember that $\hat{y}_j = 0$ if and only if $\tilde{\lambda}_j = v_j$. Otherwise we have $\hat{y}_j = 1$. Thus, the contribution of $\hat{y}_j$ to $g(\tilde{\lambda})$ is exactly $(1-\hat{y}_j)v_j + \tilde{\lambda}_j\hat{y}_j = \tilde{\lambda}_j$. As we have seen in Lemma 3.5 for a fixed k, the contribution of all $\hat{x}_{jk}$ to $g(\tilde{\lambda})$ is exactly $(1-\alpha)l_k \sum_{j \in \varphi(k)} \hat{s}_j^{\alpha}$. Summing over all k, we get that the total contribution of the $\hat{x}$-variables equals

$$\begin{aligned} \sum_{k=1}^{N}(1-\alpha)l_k \sum_{j \in \varphi(k)} \hat{s}_j^{\alpha} &= (1-\alpha)\sum_{k=1}^{N}\sum_{j \in \varphi(k)} l_k\hat{s}_j^{\alpha} = (1-\alpha)\sum_{j \in J}\sum_{k \in \varphi^{-1}(j)}^{N} l_k\hat{s}_j^{\alpha} \\ &= (1-\alpha)\sum_{j \in J} l(j)\hat{s}_j^{\alpha} = (1-\alpha)\sum_{j \in J} E_{\tilde{\lambda}}(j). \end{aligned} \tag{3.14}$$

Summing up the contributions of the $\hat{x}$- and $\hat{y}$-variables we get the desired statement. □

Tracing a Job Given a job j, we define its *trace* as a set of tuples (T_k, i) with $k \in \{1, 2, \dots, N\}$ and $i \in \{1, 2, \dots, m\}$. That is, a set of atomic intervals, each coupled with a certain processor. Our goal is to choose these such that we can account the energy $E_{\tilde{\lambda}}(j)$ used in the optimal infeasible solution on job j to the energy used by algorithm PD during j's trace (on the coupled processors).

For the formal definition, let us first partition the contributing jobs $\varphi(k)$ of an interval T_k into the subset $\varphi_1(k) := \{\, j \in \varphi(k) \mid \tilde{y}_j = 1 \,\}$ of jobs finished by PD and the subset $\varphi_2(k) := \{\, j \in \varphi(k) \mid \tilde{y}_j = 0 \,\}$ of jobs unfinished by PD. Now, for any job $j \in J$ we define its *trace* $\mathrm{Tr}(j)$ as follows[3]:

Case 1: $\tilde{y}_j = 1$

$$(T_k, i) \in \mathrm{Tr}(j) \iff \hat{s}_j \text{ is } i\text{-th largest value in } \{\, \hat{s}_{j'} \mid j' \in \varphi_1(k) \,\}$$

Case 2: $\tilde{y}_j = 0$

$$(T_k, |\varphi_1(k)| + i) \in \mathrm{Tr}(j) \iff \hat{s}_j \text{ is } i\text{-th largest value in } \{\, \hat{s}_{j'} \mid j' \in \varphi_2(k) \,\}$$

That is, jobs that are finished by PD are mapped to the fastest processors in each atomic interval T_k for which they are contributing jobs, in decreasing order of their $\hat{s}_j$-values. Jobs contributing to T_k but which are unfinished by PD are mapped to the remaining processors (the exact order is not important in this case). Note that by this mapping, all traces $\mathrm{Tr}(j)$ are pairwise disjoint. We use the notation $E_{\mathrm{PD}}(j)$ to refer to the power consumption of PD during j's trace. That is, the power consumption on the i-th fastest processor in the atomic interval T_k for any $(T_k, i) \in \mathrm{Tr}(j)$. We use E_{PD} to denote the total power consumption of PD. Since the job traces are pairwise disjoint, we obviously have $E_{\mathrm{PD}} \geq \sum_{j \in J} E_{\mathrm{PD}}(j)$.

The following proposition formulates an important structural property of a job's trace. It gives us different lower bounds on the speed used by PD during a job's trace, depending on whether it is finished or not. To this end, let $\tilde{s}_j$ denote the speed PD planned to use for job j just before $\tilde{\lambda}_j$ got fixed (i.e., just before PD decides whether to finish j or not). If j is finished, we have (cf. algorithm description and Proposition 3.1)

$$\tilde{\lambda}_j = \delta \frac{\mathrm{d}\,\mathcal{P}_k}{\mathrm{d}x_{jk}}(\tilde{x}_{1k}, \dots, \tilde{x}_{jk}, 0, \dots, 0) = \delta w_j \, \mathrm{P}'_\alpha\big(\tilde{s}_j\big). \tag{3.15}$$

Solving this for $\tilde{s}_j$ yields $\tilde{s}_j = \left(\tilde{\lambda}_j/\delta\alpha w_j\right)^{1/\alpha-1} = \delta^{-1/\alpha-1}\hat{s}_j$. Similarly, we also get $\tilde{s}_j = \delta^{-1/\alpha-1}\hat{s}_j$ for unfinished jobs. We use $\check{x}_j = \sum \check{x}_{jk} < 1$ to denote the corresponding

[3]Ties are resolved arbitrarily but consistently.

portions of the unfinished job j planned to be scheduled by PD just before j was rejected.

Proposition 3.7. *Consider $(T_k, i) \in \mathrm{Tr}(j)$ for a job $j \in J$. Let $s(i,k)$ denote the speed of the i-th fastest processor during T_k in the final schedule computed by PD. Then:*

(a) If j is finished by PD, then $s(i,k) \geq \tilde{s}_j$.

(b) If j is not finished by PD, then $s(i,k) \geq \tilde{s}_j - \frac{\check{x}_{jk} w_j}{l_k}$.

Proof.

(a) Remember that $\tilde{s}_j = \delta^{-1/\alpha - 1} \hat{s}_j$. Because of this relation and the definition of $(T_k, i) \in \mathrm{Tr}(j)$, we must have that $\tilde{s}_j$ is the i-th largest value in $\{ \tilde{s}_{j'} \mid j' \in \varphi_1(k) \}$. Together with Lemma 3.5(c), we even have that $\tilde{s}_j$ is the i-th largest value under all available jobs finished by PD. At the time τ_{k-1} (the start of interval T_k), all these available jobs j' have arrived. We consider two cases: If j is a dedicated job at this time, it is scheduled with a speed of exactly $\tilde{s}_j$. Moreover, all the $i-1$ available jobs j' with $\tilde{s}_{j'} \geq \tilde{s}_j$ are dedicated jobs and are scheduled with a speed of $\tilde{s}_{j'}$, respectively. Thus, j is scheduled on the i-th fastest processor, yielding $s(i,k) \geq \tilde{s}_j$. If j is a pool job at this time, it is scheduled on one of the pool processors at a speed of $\tilde{s}_j$ or higher. But then, since pool processors are the slowest among all processors, the i-th fastest processor must also run at a speed of at least $\tilde{s}_j$.

(b) Remember that $\check{x}_{jk}$ denotes the portion of job j PD planned to schedule in T_k just before j got rejected. If j was planned as a dedicated job, we have $l_k \tilde{s}_j = \check{x}_{jk} w_j$. This trivially yields the desired statement because of $s(i,k) \geq 0$. If j was not planned as a dedicated job, it was to be processed on a pool processor. Let $L(i,k)$ denote the workload on the i-th fastest processor during T_k just after j was rejected (i.e., without $\check{x}_{jk} w_j$). Similarly, let $L'(i,k)$ denote the workload on the i-th fastest processor during T_k just before j was rejected (i.e., including $\check{x}_{jk} w_j$). Proposition 3.2 gives us $L'(i,k) - L(i,k) \leq \check{x}_{jk} w_j$. Moreover, since j was planned as a pool job (which run at minimal speed), we must have $l_k \tilde{s}_j \leq L'(i,k)$. Combining these inequalities yields that the speed $L(i,k)/l_k$ on the i-th fastest processor during T_k at j's arrival was at least $\tilde{s}_j - \frac{\tilde{x}_{jk} w_j}{l_k}$. As Proposition 3.2 also implies that the workload (and, thus, the speed) of the i-th fastest processor in an atomic interval can only increase due to the arrival of new jobs, we get the desired statement. □

3.4.3 Balancing the Different Cost Components

As our goal is to lower-bound the dual function value $g(\tilde{\lambda}) = (1-\alpha)\sum E_{\tilde{\lambda}}(j) + \sum \tilde{\lambda}_j$ by the cost of algorithm PD, we have to relate the values $E_{\tilde{\lambda}}(j)$ and $\tilde{\lambda}_j$ to the energy- and value- costs of PD. It depends on the job itself how this is done exactly. For example, in the case of finished jobs, both terms can be related to the actual energy consumption of PD in a relatively straightforward way. This becomes much harder if the job is not finished by PD: after all, in this case PD does not invest any energy into the job. The job's trace plays a crucial role in this case, as it allows us to account the energy investment of the optimal infeasible solution to the energy PD consumed during the trace. The next proposition gathers the most important relations to be used in the following proofs.

Proposition 3.8. *Consider an arbitrary job $j \in J$. The following properties hold:*

(a) $E_{\tilde{\lambda}}(j) = \tilde{\lambda}_j \frac{\hat{x}_j}{\alpha}$.

(b) If j is finished by PD, then $E_{\tilde{\lambda}}(j) \leq \delta^{\frac{\alpha}{\alpha-1}} E_{PD}(j)$.

(c) If j is not finished by PD and $\hat{x}_j > \delta^{\frac{1}{\alpha-1}}$, then

$$E_{\tilde{\lambda}}(j) < \delta^{\frac{\alpha}{\alpha-1}} \left(1 - \frac{\delta^{\frac{1}{\alpha-1}}}{\hat{x}_j}\right)^{-\alpha} E_{PD}(j). \tag{3.16}$$

Proof.

(a) By using the identities $\hat{s}_j = \left(\tilde{\lambda}_j/\alpha w_j\right)^{1/\alpha - 1}$ and $l(j)\hat{s}_j = \hat{x}_j w_j$ (see Lemma 3.5) we can compute

$$E_{\tilde{\lambda}}(j) = l(j)\hat{s}_j^{\alpha} = l(j)\hat{s}_j \cdot \hat{s}_j^{\alpha-1} = \hat{x}_j w_j \cdot \frac{\tilde{\lambda}_j}{\alpha w_j} = \tilde{\lambda}_j \frac{\hat{x}_j}{\alpha}. \tag{3.17}$$

(b) Assume j is finished by PD. Remember that $\tilde{s}_j$ denotes the speed assigned to j when it arrived and $\tilde{\lambda}_j$ got fixed. We have the relation $\tilde{s}_j = \delta^{-1/\alpha-1}\hat{s}_j$ (cf. Section 3.4.2). Let $s_{\min}$ denote the minimal speed of j's trace in the final $(\tilde{x}, \tilde{y})$-schedule produced by PD. That is, there is a tuple $(T_k, i) \in \mathrm{Tr}(j)$ such that the i-th fastest processor in T_k runs at speed $s_{\min}$ and $E_{\mathrm{PD}}(j) \geq l(j)s_{\min}^{\alpha}$. By Proposition 3.7 we must have $s_{\min} \geq \tilde{s}_j$. We compute

$$E_{\tilde{\lambda}}(j) = l(j)\hat{s}_j^{\alpha} = \delta^{\frac{\alpha}{\alpha-1}} l(j)\tilde{s}_j^{\alpha} \leq \delta^{\frac{\alpha}{\alpha-1}} l(j)s_{\min}^{\alpha} \leq \delta^{\frac{\alpha}{\alpha-1}} E_{\mathrm{PD}}(j). \tag{3.18}$$

(c) Applying Proposition 3.7 to all $(T_k, i) \in \mathrm{Tr}(j)$ yields that the total workload L that is processed by PD during j's trace is at least $l(j)\tilde{s}_j - \tilde{x}_j w_j > l(j)\tilde{s}_j - w_j$. The minimum energy necessary to process this workload in $l(j)$ time units is $l(j)(L/l(j))^{\alpha}$. We compute

$$\begin{aligned} E_{\mathrm{PD}}(j) &\geq l(j)\left(\frac{L}{l(j)}\right)^{\alpha} > l(j)\left(\frac{l(j)\tilde{s}_j - w_j}{l(j)}\right)^{\alpha} = l(j)\tilde{s}_j^{\alpha}\left(1 - \frac{w_j}{\tilde{s}_j l(j})\right)^{\alpha} \\ &= \delta^{-\frac{\alpha}{\alpha-1}} E_{\tilde{\lambda}}(j)\left(1 - \frac{\delta^{\frac{1}{\alpha-1}}}{\hat{x}_j}\right)^{\alpha}. \end{aligned} \tag{3.19}$$

Rearranging the inequality yields the desired statement. □

Note that the bound for unfinished jobs in Proposition 3.8 has an additional factor > 1 compared to the one for finished jobs. However, for large enough $\hat{x}_j$ this factor becomes nearly one. Thus, we will apply this bound only in cases of large $\hat{x}_j$. If $\hat{x}_j$ is relatively small, we will instead bound $E_{\tilde{\lambda}}(j)$ only by its value. We continue by describing the different types of jobs we consider. In total, we differentiate between three job categories:

Finished Jobs These are all jobs j with $\tilde{y}_j = 1$ (i.e., jobs finished by PD). As mentioned above, we bound both components $E_{\tilde{\lambda}}(j)$ and $\tilde{\lambda}_j$ of $g(\tilde{\lambda})$ by the actual energy consumption of PD. We use

$$J_1 := \left\{\, j \in J \,\middle|\, \tilde{y}_j = 1 \,\right\} \tag{3.20}$$

to refer to this job category.

Unfinished, Low-yield Jobs We use the term *low-yield jobs* to refer to jobs not finished by PD and which have a relatively small $\hat{x}_j$. That is, jobs of which the optimal infeasible solution does not schedule too large a portion. Intuitively, the value of such jobs must be small, because otherwise it would have been beneficial to schedule a larger portion of them in the optimal infeasible solution. In this sense, these jobs are low-yield and we will exploit this fact by bounding both components $E_{\tilde{\lambda}}(j)$ and $\tilde{\lambda}_j$ of $g(\tilde{\lambda})$ by the job value PD is charged for not finishing j. More formally, this job category is defined as

$$J_2 := \left\{ j \in J \,\middle|\, \tilde{y}_j = 0 \wedge \hat{x}_j \leq \frac{\alpha - \alpha^{1-\alpha}}{\alpha - 1} \right\}. \tag{3.21}$$

Unfinished, High-yield Jobs Correspondingly, the term *high-yield jobs* refers to jobs finished by PD and which have a relatively large $\hat{x}_j$. More exactly, these jobs are given by

$$J_3 := \left\{ j \in J \,\middle|\, \tilde{y}_j = 0 \wedge \hat{x}_j > \frac{\alpha - \alpha^{1-\alpha}}{\alpha - 1} \right\}. \tag{3.22}$$

This proves to be the most challenging case, as neither do the jobs feature a particularly small value nor does PD invest any energy into their execution. Instead, we use a mix of the job's value and the energy spent by PD during j's trace to account for its contribution. One has to carefully balance what portions of $E_{\tilde{\lambda}}(j)$ and $\tilde{\lambda}_j$ to bound by either $E_{\mathrm{PD}}(j)$ or by v_j.

In accordance with these job categories, we split the value of the dual function by the corresponding contributions. That is, $g(\tilde{\lambda}) = \sum_{i=1}^{3} g_i(\tilde{\lambda})$, where $g_i(\tilde{\lambda}) = (1-\alpha)\sum_{j \in J_i} E_{\tilde{\lambda}}(j) + \sum_{j \in J_i} \tilde{\lambda}_j$. The following lemmas bound each contribution separately.

Lemma 3.9 (Finished Jobs). *For finished jobs it holds that*

$$g_1(\tilde{\lambda}) \geq \delta E_{PD} + (1-\alpha)\delta^{\frac{\alpha}{\alpha-1}} \sum_{j \in J_1} E_{PD}(j). \tag{3.23}$$

Proof. We have $g_1(\tilde{\lambda}) = (1-\alpha)\sum_{j \in J_1} E_{\tilde{\lambda}}(j) + \sum_{j_1 \in J} \tilde{\lambda}_j$. Using Proposition 3.8(b) and $\alpha > 1$ we bound the first summand by $(1-\alpha)\delta^{\frac{\alpha}{\alpha-1}} \sum_{j \in J_1} E_{\mathrm{PD}}(j)$. For the second summand, we get

$$\begin{aligned}\sum_{j\in J_1}\tilde{\lambda}_j &= \sum_{j\in J_1}\sum_{k=1}^{N}\tilde{x}_{jk}\tilde{\lambda}_j = \sum_{j\in J_1}\sum_{k=1}^{N}\tilde{x}_{jk}\delta\frac{\mathrm{d}\,\mathcal{P}_k}{\mathrm{d}x_{jk}}(\tilde{x}_{1k},\dots,\tilde{x}_{jk},0,\dots,0)\\ &= \delta\sum_{k=1}^{N}\sum_{j\in J}\tilde{x}_{jk}\frac{\mathrm{d}\,\mathcal{P}_k}{\mathrm{d}x_{jk}}(\tilde{x}_{1k},\dots,\tilde{x}_{jk},0,\dots,0)\\ &\geq \delta\sum_{k=1}^{N}\mathcal{P}_k(\tilde{x}_{1k},\dots,\tilde{x}_{nk}) = \delta E_{\mathrm{PD}}.\end{aligned} \tag{3.24}$$

The involved inequality is based on the fact that for any differentiable convex function $f\colon \mathbb{R}^n \to \mathbb{R}$ with $f(0) = 0$ and $x \in \mathbb{R}^n_{\geq 0}$ we have the inequality

$$\sum_{j=1}^{n} x_j\frac{\mathrm{d}f}{\mathrm{d}x_j}(x_1,\dots,x_j,0,\dots,0) \geq f(x) \tag{3.25}$$

(see, e.g., [BV04, Chapter 3]). Combining these bounds yields the lemma's statement. □

Lemma 3.10 (Low-yield Jobs). *For unfinished, low-yield jobs it holds that*

$$g_2(\tilde{\lambda}) \geq \alpha^{-\alpha}\sum_{j\in J_2} v_j. \tag{3.26}$$

Proof. Proposition 3.8(a) together with the fact that $\tilde{\lambda}_j = v_j$ for $j \in J_2$ yields $E_{\tilde{\lambda}}(j) = v_j\frac{\hat{x}_j}{\alpha}$. Applying this to $g_2(\tilde{\lambda})$ we get

$$\begin{aligned}g_2(\tilde{\lambda}) &= \sum_{j\in J_2}(1-\alpha)E_{\tilde{\lambda}}(j) + \sum_{j\in J_2}\tilde{\lambda}_j = \sum_{j\in J_2}\frac{1-\alpha}{\alpha}\hat{x}_j v_j + \sum_{j\in J_2} v_j\\ &= \sum_{j\in J_2}\left(1-\frac{\alpha-1}{\alpha}\hat{x}_j\right)v_j \overset{\text{Def.}}{\underset{J_2}{\geq}} \sum_{j\in J_2}\left(1-\frac{\alpha-\alpha^{1-\alpha}}{\alpha}\right)v_j\\ &= \alpha^{-\alpha}\sum_{j\in J_2} v_j.\end{aligned}$$ □

Lemma 3.11 (High-yield Jobs). *For unfinished, high-yield jobs and if $\delta \leq \frac{1}{\alpha^{\alpha-1}}$, it holds that*

$$g_3(\tilde{\lambda}) \geq \frac{1-\alpha}{\alpha^{\alpha}}\sum_{j\in J_3} E_{PD}(j) + \alpha^{-\alpha}\sum_{j\in J_3} v_j. \tag{3.27}$$

Proof. We make use of both Proposition 3.8(a) and Proposition 3.8(c). First note that the prerequisite $\delta \leq 1/\alpha^{\alpha-1}$ together with $\alpha > 1$ and $j \in J_3$ gives us the relation $\delta^{\frac{1}{\alpha-1}} \leq \frac{1}{\alpha} \leq 1 \leq \frac{\alpha-\alpha^{1-\alpha}}{\alpha-1} < \hat{x}_j$. This allows us to apply Proposition 3.8(c). The second summand of $g_3(\tilde{\lambda})$ is split into two parts, one of which is accounted for by energy invested by PD and the other one by lost value due to unfinished jobs:

$$\begin{aligned}
g_3(\tilde{\lambda}) &= \sum_{j\in J_3}(1-\alpha)E_{\tilde{\lambda}}(j) + \sum_{j\in J_3}\tilde{\lambda}_j \\
&= \sum_{j\in J_3}(1-\alpha)E_{\tilde{\lambda}}(j) + \sum_{j\in J_3}(1-\alpha^{-\alpha})\tilde{\lambda}_j + \sum_{j\in J_3}\alpha^{-\alpha}\tilde{\lambda}_j \\
&= \sum_{j\in J_3}(1-\alpha)E_{\tilde{\lambda}}(j) + \sum_{j\in J_3}(1-\alpha^{-\alpha})\frac{\alpha E_{\tilde{\lambda}}(j)}{\hat{x}_j} + \sum_{j\in J_3}\alpha^{-\alpha}v_j \\
&= \sum_{j\in J_3}(1-\alpha)E_{\tilde{\lambda}}(j)\left(1 - \frac{\alpha-\alpha^{1-\alpha}}{(\alpha-1)\hat{x}_j}\right) + \sum_{j\in J_3}\alpha^{-\alpha}v_j \\
&> \sum_{j\in J_3}(1-\alpha)\delta^{\frac{\alpha}{\alpha-1}}E_{\mathrm{PD}}(j)\left(1-\frac{\delta^{\frac{1}{\alpha-1}}}{\hat{x}_j}\right)^{-\alpha}\left(1 - \frac{\alpha-\alpha^{1-\alpha}}{(\alpha-1)\hat{x}_j}\right) + \sum_{j\in J_3}\alpha^{-\alpha}v_j \\
&\geq \sum_{j\in J_3}(1-\alpha)\alpha^{-\alpha}\left(1-\frac{1}{\alpha\hat{x}_j}\right)^{-\alpha}\left(1-\frac{1}{\hat{x}_j}\right)E_{\mathrm{PD}}(j) + \sum_{j\in J_3}\alpha^{-\alpha}v_j \\
&\geq (1-\alpha)\alpha^{-\alpha}\sum_{j\in J_3}E_{\mathrm{PD}}(j) + \sum_{j\in J_3}\alpha^{-\alpha}v_j.
\end{aligned}$$

The first inequality applies Proposition 3.8(c), the penultimate inequality the relations deduced from the prerequisite, and the last inequality is the application of Bernoulli's inequality. □

3.4.4 Deriving the Tight Competitive Ratio

It remains to derive our final upper bound on the competitive ratio of PD. We do so by combining the bounds from Lemma 3.9, Lemma 3.10, and Lemma 3.11.

Theorem 3.3. *The competitive ratio of algorithm PD with the parameter δ set to $1/\alpha^{\alpha-1}$ is at most α^α. Moreover, there is a problem instance for which PD is exactly by a factor of α^α worse than an optimal algorithm. That is, our upper bound is optimal.*

Proof. If we combine the results from Lemma 3.9 to Lemma 3.11 we get

$$\begin{aligned}
g(\tilde{\lambda}) &\geq \alpha^{1-\alpha} E_{\mathrm{PD}} + (1-\alpha)\alpha^{-\alpha} \sum_{j \in J_1 \cup J_3} E_{\mathrm{PD}}(j) + \alpha^{-\alpha} \sum_{j \in J_2 \cup J_3} v_j \\
&\geq \alpha^{1-\alpha} E_{\mathrm{PD}} + (1-\alpha)\alpha^{-\alpha} \sum_{j \in J} E_{\mathrm{PD}}(j) + \alpha^{-\alpha} \sum_{j \in J_2 \cup J_3} v_j \\
&\geq \bigl(\alpha^{1-\alpha} + (1-\alpha)\alpha^{-\alpha}\bigr) E_{\mathrm{PD}} + \alpha^{-\alpha} \sum_{j \in J_2 \cup J_3} v_j \\
&= \alpha^{-\alpha} \operatorname{cost}(\mathrm{PD}).
\end{aligned} \tag{3.28}$$

Now, let OPT denote an optimal schedule for the current problem instance. Moreover, let OPT$'$ denote an optimal solution to the relaxed mathematical program (CP). Obviously, it holds that $\operatorname{cost}(\mathrm{OPT}') \leq \operatorname{cost}(\mathrm{OPT})$. By duality, we know that $g(\tilde{\lambda}) \leq \operatorname{cost}(\mathrm{OPT}')$. By combining these inequalities we can bound PD's competitiveness by

$$\operatorname{cost}(\mathrm{PD}) \leq \alpha^{\alpha} g(\tilde{\lambda}) \leq \alpha^{\alpha} \operatorname{cost}(\mathrm{OPT}') \leq \alpha^{\alpha} \operatorname{cost}(\mathrm{OPT}). \tag{3.29}$$

For the lower bound, consider a single processor and assume the job values are high enough to ensure that PD finishes all jobs. We create a job instance of n jobs in the same way as done in [BKP04] for the lower bound on OA and AVR. That is, job $j \in J = \{1, 2, \ldots, n\}$ arrives at time $j - 1$ and has workload $(n-j+1)^{-1/\alpha}$. All jobs have the same deadline n. Now, whenever one of the jobs arrives, PD schedules all remaining jobs at the energy-optimal (i.e., minimal) speed as pool jobs. In other words, it computes a schedule that is optimal for the remaining known work. This is exactly what OA does (hence its name), which means that we get the same lower bound of α^{α} as for OA (cf. [BKP04, Lemma 3.2]). □

3.5 Conclusion & Outlook

This chapter presented a new algorithm and an analysis based on duality theory for scheduling valuable jobs on multiple speed-scalable processors. Using duality theory for the analysis of energy-efficient scheduling algorithms was proposed by Gupta et al. [GKP12]. Given that the first formal proof of the original offline algorithm's optimality was achieved by means of duality theory using the KKT conditions [BKP07], it seems that this is a very natural way to approach this kind of problems. However, almost all results for online algorithms in this area use amortized competitiveness arguments similar to the original proof of OA's competitiveness, one of the first and most important online algorithms for energy-efficient scheduling. While this approach proved to be elegant and very powerful, designing suitable potential functions is difficult and needs a quite high amount of experience with the topic. Adapting these potential functions to new model variations and generalizations, or tuning them to narrow the gap to the known lower bounds, is non-trivial and remains a challenging task. Using well-developed utilities from duality theory for convex programming may prove to be a worthwhile alternative approach. After all, this approach allowed us not only to improve upon known results but also to generalize them to the important case of multiple processors. Nguyen [Ngu13] indicates that such duality-based methods seem to be the "right" tool to analyze speed scaling and resource augmentation models.

However, currently the usage of such convex programming techniques for online algorithms in speed scaling literature is limited to algorithms that are, in some sense, greedy. This is a consequence of the fact that these algorithms' design mostly follows the typical primal-dual approach [BN09], where primal and/or dual variables are greedily raised until certain conditions are met. In the setting of speed scaling, such greedy algorithms seem to be quite limited. For example, to the best of my knowledge, all known greedy algorithms for deadline scheduling have a competitiveness of at least α^{α}. It is not obvious how to conduct a duality-based analysis on non-greedy algorithms. The algorithm qOA might be a good candidate for a first try: its potential function based analysis is not tight and seems relatively weak.

CHAPTER 4

Slow Down & Sleep for Profit

“ *Death was Nature's way of telling you to slow down.* ”

Terry Pratchett, Strata

The ubiquity of technical systems, the rise of mobile computing, a growing ecological awareness in the public; all of these are reasons why energy efficiency has become a major concern in our society. The previous chapters already discussed the importance of energy saving techniques, especially speed scaling. In combination with improvements on the technical level, algorithmic research has great potential to reduce energy consumption. Albers [Alb10] gives a good insight into the role of algorithms to fully exploit the diverse energy-saving opportunities.

Besides speed scaling, power-down mechanisms are another prominent technique to save energy. While the former allows a system to save energy by adapting the processor's speed to the current system load, the latter can be used to transition into a sleep mode to conserve energy. In the following, we are interested in the combination of both mechanisms: speed scaling with a (single) sleep state on one processor. From an algorithmic viewpoint, the most challenging aspect in the design of scheduling strategies for this setting is to handle the lack of knowledge about the future: should we use a high speed to free resources in anticipation of new jobs or enter sleep mode in the hope that no new jobs arrive in the near future? This scenario has been considered before, see [ISG07] and the literature overview in Section 4.1. However, this is not the case for the price-collecting variant of this problem, which will be the subject of our study in this chapter.

Chapter Basis The model, analysis, and results presented in the remainder of this chapter are based on the following publication:

> **2012** (with A. Cord-Landwehr and F. Mallmann-Trenn). "Slow Down and Sleep for Profit in Online Deadline Scheduling". In: *Proceedings of the 1st Mediterranean Conference on Algorithms (MedAlg)*, cf. [CKM12].

Chapter Outline We start with an overview of related literature in Section 4.1, and use this to put the results of this chapter into perspective. Section 4.2 presents the formal model description, the necessary notation, and some basic facts about the structure of certain schedules. A first small result is stated and proven in Section 4.3: a lower bound on the competitiveness for a class of algorithms which we refer to as *rejection-oblivious* algorithms. Our main result follows in Section 4.4, where we present our algorithm and its analysis. Section 4.5 closes the chapter's analysis part with a short discussion of the setting with a bounded maximum speed. We conclude the chapter with a short résumé in Section 4.6.

4.1 Related Work & Contribution

As for the previous chapter, we provide a self-contained literature overview, even if that means the reader may have to bear with a slight repetition. In the present chapter, we focus on theoretical results concerned with deadline scheduling on a single, speed-scalable processor and with sleep states.

Theoretical work in this area has been initiated by Yao et al. [YDS95]. They considered scheduling of jobs having different sizes and deadlines on a single variable-speed processor. When running at speed s, its power consumption is

$P(s) = s^\alpha$ for some constant $\alpha \geq 2$. Yao et al. derived an optimal polynomial-time offline algorithm as well as two online algorithms known as *optimal available* (OA) and *average rate* (AVR). Up to now, OA remains one of the most important algorithms in this area, as it is used as a basic building block by many strategies (including the strategy we present in this chapter). Using an elegant amortized potential function argument, Bansal et al. [BKP07] were able to show that OA's competitive ratio is exactly α^α. Moreover, the authors stated a new algorithm, named BKP, which achieves a competitive ratio of essentially $2e^{\alpha+1}$. This improves upon OA for large α. The best known lower bound for deterministic algorithms is $e^{\alpha-1}/\alpha$ due to Bansal et al. [Ban+09]. They also presented an algorithm (qOA) that is particularly well-suited for low powers of α. An interesting and realistic model extension is the restriction of the maximum processor speed. In such a setting, a scheduler may not always be able to finish all jobs by their deadlines. Chan et al. [Cha+07] were the first to consider the combination of classical speed scaling with such a maximum speed. They gave an algorithm that is $\alpha^\alpha + \alpha^2 4^\alpha$-competitive on energy and 14-competitive on throughput. Bansal et al. [Ban+08b] improved this to a 4-competitive algorithm concerning the throughput while maintaining a constant competitive ratio with respect to the energy. Note that no algorithm – even if ignoring the energy consumption – can be better than 4-competitive for throughput (see [Bar+91]).

Power-down mechanisms were studied by Baptiste [Bap06]. He considered a fixed-speed processor requiring a certain amount of energy to stay awake, but which may switch into a sleep state to save energy. Returning from sleep needs energy γ. For jobs of unit size, he gave an optimal polynomial-time offline algorithm, which was later extended to jobs of arbitrary size [BCD07]. The first work to combine both dynamic speed scaling and sleep states in the classical YAO-model is due to Irani et al. [ISG07]. They achieved a 2-approximation for arbitrary convex power functions. For the online setting and power function $P(s) = s^\alpha + \beta$ a competitive ratio of $4^{\alpha-1}\alpha^\alpha + 2^{\alpha-1} + 2$ was reached. Han et al. [Han+10] improved upon this in two respects: they lowered the competitive ratio to $\alpha^\alpha + 2$ and transferred the result to scenarios limiting the maximum speed. Only recently, Albers and Antoniadis [AA12] proved that the optimization problem is NP-hard and gave lower bounds for several algorithm classes. Moreover, they improved the approximation ratio for general convex power functions to $4/3$. The papers most closely related to this work are due to Pruhs and Stein [PS10] and Chan et al. [CLL10a]. Both considered the dynamic speed scaling model of Yao et al. However, they extended the idea of energy-minimal schedules to a profit-oriented objective. In the simplest case, jobs have values (or priorities) and the scheduler is no longer required to finish all jobs. Instead, it can decide to reject jobs whose values do not justify the foreseeable energy investment necessary to complete them. The objective is to maximize profit [PS10] or, similarly, minimize the loss [CLL10a]. As argued by the authors, the latter model has the benefit of being a direct generalization of the classical model of Yao et al. [YDS95]. For maximizing the profit, Pruhs and Stein [PS10] showed that, in order to achieve a bounded competitive ratio, resource augmentation is necessary and designed a scalable online algorithm. For minimizing the loss, Chan et al. [CLL10a] gave a $\alpha^{\alpha} + 2e\alpha$-competitive algorithm and transferred the result to the case of a bounded maximum speed.

Our Contribution This chapter presents the first model that not only takes into account two of the most prominent energy saving techniques (namely, speed scaling and power-down) but couples the energy minimization objective with the idea of profitability. It combines aspects from both [ISG07] and [CLL10a]. From [ISG07] it inherits one of the most realistic processor models considered in this area: a single variable-speed processor with power function $P(s) = s^{\alpha} + \beta$ and a sleep state. Thus, even at speed zero the system is charged a certain amount β of energy, but it can suspend to sleep such that no energy is consumed. Waking up causes a transition cost of γ. The job model stems from [CLL10a]: Jobs arrive in an online fashion, are preemptable, and have a deadline, size, and value. The scheduler can reject jobs (e.g., if their values do not justify the presumed energy investment). Its objective is to minimize the total energy investment plus the total value of rejected jobs.

One of this chapter's major insights is that the maximum value density $\delta_{\max}$ (i.e., the ratio between a job's value and its workload) is a parameter that is inherently connected to the necessary and sufficient competitive ratio achievable for the considered online scheduling problem. We present an online algorithm that combines ideas from [CLL10a] and [Han+10] and analyze its competitive ratio with respect to $\delta_{\max}$. This yields an upper bound of

$\alpha^\alpha + 2e\alpha + \delta_{\max}\frac{s_{\mathrm{cr}}}{P(s_{\mathrm{cr}})}$.[1] If the value density of low-valued jobs is not too large or job values are at least γ, the competitive ratio becomes $\alpha^\alpha + 2e\alpha$. Moreover, we show that one cannot do much better: any *rejection-oblivious* strategy has a competitive ratio of at least $\delta_{\max}\frac{s_{\mathrm{cr}}}{P(s_{\mathrm{cr}})}$. Here, rejection-oblivious means that rejection decisions are based on the *current* system state and job properties only. This lower bound is in stark contrast to the setting without sleep states, where a rejection-oblivious $\mathrm{O}(1)$-competitive algorithm exists [CLL10a]. Using the definition of a job's penalty ratio (due to Chan et al. [CLL10a]), we extend our results to processors with a bounded maximum speed.

4.2 Model & Preliminaries

We are given a speed-scalable processor that can be set to any speed $s \in [0, \infty)$. When running at speed s its power consumption is $P_{\alpha,\beta}(s) = s^\alpha + \beta$ with $\alpha \geq 2$ and $\beta \geq 0$. If $s(t)$ denotes the processor speed at time t, the total power consumption is $\int_0^\infty P_{\alpha,\beta}(s(t))\,\mathrm{d}t$. We can suspend the processor into a sleep state to save energy. In this state, it cannot process any jobs and has a power consumption of zero. Though entering the sleep state is free of costs, waking up needs a fixed *transition energy* $\gamma \geq 0$. Over time, n jobs $J = \{1, 2, \dots, n\}$ are released. Each job j appears at its release time r_j and has a deadline d_j, a (non-negative) value v_j, and requires a certain amount w_j of work. The processor can process at most one job at a time. Preemption is allowed, i.e., jobs may be paused at any time and continued later on. If I denotes the period of time (not necessarily an interval) when j is scheduled, the amount of work processed is $\int_I s(t)\,\mathrm{d}t$. A job is finished if $\int_I s(t)\,\mathrm{d}t \geq w_j$. Jobs not finished until their deadline cause a cost equal to their value. We call such jobs *rejected*. A schedule S specifies for any time t the processor's state (asleep or awake), the currently processed job (if the processor is awake), and sets the speed $s(t)$. W.l.o.g., we assume $s(t) = 0$ when no job is being processed. Initially, the processor is assumed to be asleep. Whenever it is neither sleeping nor working we say it is *idle*. A schedule's cost is the invested energy (for awaking from sleep, idling, and working on jobs) plus the loss due to rejected jobs. Let m denote the number of sleep intervals, l the total length of idle intervals, and

[1]The expression $\frac{s_{\mathrm{cr}}}{P(s_{\mathrm{cr}})}$ depends only on α and β, see Section 4.2.

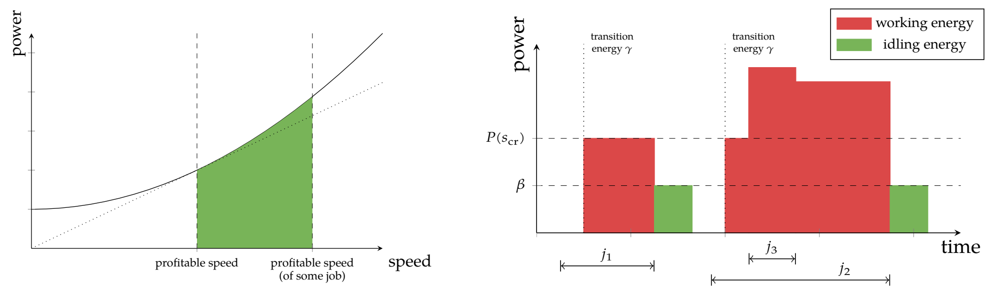
$\mathcal{I}_{\text{work}}$ the collection of all working intervals (i.e., times when $s(t) > 0$). Then the schedule's *sleeping energy* is $E^S_{\text{sleep}} := (m-1)\gamma$, its *idling energy* is $E^S_{\text{idle}} := l\beta$, and its *working energy* is $E^S_{\text{work}} := \int_{\mathcal{I}_{\text{work}}} P_{\alpha,\beta}(s(t))\,\mathrm{d}t$. We use V^S_{rej} to denote the total value of rejected jobs. Now, the cost of schedule S is

$$\operatorname{cost}(S) := E^S_{\text{sleep}} + E^S_{\text{idle}} + E^S_{\text{work}} + V^S_{\text{rej}}. \tag{4.1}$$

We seek online strategies yielding a provably good schedule. More formally, we measure the quality of online strategies by their competitive ratio: For an online algorithm A and a problem instance I let $A(I)$ denote the resulting schedule and $O(I)$ an optimal schedule for I. Then A is said to be c-competitive if $\sup_I \frac{\operatorname{cost}(A(I))}{\operatorname{cost}(O(I))} \leq c$.

We define the *system energy* E^S_{sys} of a schedule to be the energy needed to hold the system awake (whilst idling and working). That is, if S is awake for a total of x time units, $E^S_{\text{sys}} = x\beta$. Note that $E^S_{\text{sys}} \leq E^S_{\text{idle}} + E^S_{\text{work}}$. The *critical speed* of the power function is defined as $s_{\text{cr}} := \arg\min_{s \geq 0} {}^{P_{\alpha,\beta}(s)}/_{s}$ (cf. also [ISG07; Han+10]). If job j is processed at constant speed s its energy usage is $w_j \cdot {}^{P_{\alpha,\beta}(s)}/_{s}$. Thus, assuming that j is the only job in the system and ignoring its deadline, s_{cr} is the energy-optimal speed to process j. One can easily check that $s^{\alpha}_{\text{cr}} = \frac{\beta}{\alpha-1}$. Given a job j, let $\delta_j := {}^{v_j}/_{w_j}$ denote the job's *value density*. Following [CLL10a] and [PS10], we define the *profitable speed* $s_{j,\text{p}}$ of job j to be the maximum speed for which its processing may be profitable. More formally, $s_{j,\text{p}} := \max\{\, s \geq 0 \mid w_j \cdot {}^{P_{\alpha,0}(s)}/_{s} \leq v_j \,\}$. Note that the definition is with respect to $P_{\alpha,0}$, i.e., it ignores the system energy. The profitable speed can be more explicitly characterized by $s^{\alpha-1}_{j,\text{p}} = \delta_j$. It is easy to see that a schedule that processes j at an average speed faster than $s_{j,\text{p}}$ cannot be optimal: rejecting j and idling during the former execution phase would be more profitable. See Figure 4.1 for an illustration of these notions.

Optimal Available & Structural Properties One of the first online algorithms for dynamic speed scaling was Optimal Available (OA) due to [YDS95]. As it is an essential building block of not only our but many algorithms for speed scaling, we give a short recap of its idea (see [BKP07] for a thorough discussion and analysis). At any time, OA computes an optimal offline schedule assuming that no further jobs arrive. This optimal offline schedule is com-



(a) Our algorithm tries to use job speeds that essentially stay within the shaded interval.

(b) A sample schedule and the involved energy types.

Figure 4.1: Basic notions and sample schedule for speed scaling with a sleep state.

puted as follows: Let the density of an interval I be defined as $^{w(I)}/_{|I|}$. Here, $w(I)$ denotes the total work of jobs j with $[r_j, d_j) \subseteq I$ and $|I|$ the length of I. Now, whenever a job arrives OA computes so-called *critical intervals* by iteratively choosing an interval of maximum density. Jobs are then scheduled at a speed equal to the density of the corresponding critical interval using the earliest deadline first policy. Let us summarize several structural facts known about the OA schedule.

Fact 4.1. Let S and S' denote the OA schedules just before and after j's arrival. We use $S(j)$ and $S'(j)$ to denote j's speed in the corresponding schedule.

(a) The speed function of S (and S') is a non-increasing staircase function.

(b) The minimal speed of S' during $[r_j, d_j)$ is at least $S'(j)$.

(c) Let I be an arbitrary period of time during $[r_j, d_j)$ (not necessarily an interval). Moreover, let W denote the total amount of work scheduled by S and W' the one scheduled by S' during I. Then the inequality $W \leq W' \leq W + w_j$ holds.

(d) The speed of any $j' \neq j$ can only increase due to j's arrival: $S'(j') \geq S(j')$.

4.3 Lower Bound for Rejection-Oblivious Algorithms

This section considers a class of simple, deterministic online algorithms that we call *rejection-oblivious*. When a job arrives, a rejection-oblivious algorithm decides whether to accept or reject the job. This decision is based solely on the processor's current state (sleeping, idling, working), its current workload, and the job's properties. In particular, it does not take former decisions into account. An example for such an algorithm is $PS(c)$ from [CLL10a]. For a suitable parameter c, it is $\alpha^\alpha + 2e\alpha$-competitive in a model without sleep state. In this section we show that in our model (i.e., with a sleep state) no rejection-oblivious algorithm can be competitive. More exactly, the competitiveness of any such algorithm can become arbitrarily large. We identify the jobs' value density as a crucial parameter for the competitiveness of these algorithms.

Theorem 4.2. *The competitiveness of any rejection-oblivious algorithm A is unbounded. More exactly, for any A there is a problem instance I with competitive ratio $\geq \delta_{max}\frac{s_{cr}}{P_{\alpha,\beta}(s_{cr})}$. Here, δ_{max} is the maximum value density of jobs from I.*

Proof. For A to be competitive, there must be some $x \in \mathbb{R}$ such that, while A is asleep, all jobs of value at most x are rejected (independent of their work and deadlines). Otherwise, we can define a sequence of n identical jobs $1, 2, \dots, n$ of arbitrary small value ε. W.l.o.g., we release them such that A goes to sleep during $[d_{j-1}, r_j)$ (otherwise A consumes an infinite amount of energy). Thus, A's cost is at least $n\gamma$. If we instead consider a schedule S that rejects all jobs, we have $\operatorname{cost}(S) = n\varepsilon$. For $\varepsilon \to 0$ we see that A's competitive ratio is unbounded.

So, let $x \in \mathbb{R}$ be such that A rejects any job of value at most x whilst asleep. Consider n jobs of identical value x and work w. For each job, the deadline is set such that $w = s_{\mathrm{cr}}(d_j - r_j)$. The jobs are released in immediate succession, i.e., $r_j = d_{j-1}$. Algorithm A rejects all jobs, incurring cost nx. Let S denote the schedule that accepts all jobs and processes them at speed s_{cr}. The cost of S is given by $\operatorname{cost}(S) = \gamma + nw\frac{P_{\alpha,\beta}(s_{\mathrm{cr}})}{s_{\mathrm{cr}}}$. Thus, A's competitive ratio is at least

$$\frac{nx}{\gamma + nw\frac{P_{\alpha,\beta}(s_{\mathrm{cr}})}{s_{\mathrm{cr}}}} = \delta_{\max}\frac{1}{\frac{\gamma}{nw} + \frac{P_{\alpha,\beta}(s_{\mathrm{cr}})}{s_{\mathrm{cr}}}}. \tag{4.2}$$

For $n \to \infty$ we get the lower bound $\delta_{\max}\frac{s_{\mathrm{cr}}}{P_{\alpha,\beta}(s_{\mathrm{cr}})}$. □

4.4 Algorithm & Analysis

In the following, we use A to refer to both the algorithm and the schedule it produces; which is meant will be clear from the context. As most algorithms in this area (see, e.g., [ISG07; Ban+09; CLL10a; Han+10; AAG11]), A relies heavily on the good structural properties of OA and on OA's wide applicability to variants of the original energy-oriented scheduling model of Yao et al. [YDS95]. It essentially consists of two components, the *rejection policy* and the *scheduling policy*. The rejection policy decides which jobs to accept or reject, while the scheduling policy ensures that all accepted jobs are finished until their deadline. Our rejection policy is an extension of the one used by the algorithm PS in [CLL10a]. It ensures that we process only jobs that have a reasonable high value (value > planned energy investment) and that we do not wake from sleep for very cheap jobs. The scheduling policy controls the speed, the job assignment, and the current mode of the processor. It is a straightforward adaption of the algorithm used in [Han+10]. However, its analysis proves to be more involved because we have to take into account its interaction with the rejection policy and that the job sets scheduled by the optimal algorithm and A may be quite different.

The following descriptions of the scheduling and rejection policies assume a continuous recomputation of the current OA schedule. See Listing 4.1 for the corresponding pseudocode. It is straightforward to implement A such that the planned schedule is recomputed only when new jobs arrive.

Scheduling Policy All accepted jobs are scheduled according to the earliest deadline first rule. At any time, the processor speed is computed based on the OA schedule. We use OA^t to denote the schedule produced by OA if given the remaining (accepted) work at time t and the power function $P_{\alpha,0}$. Let ρ_t denote the speed planned by OA^t at time t. A puts the processor either in *working*, *idling*, or *sleeping* mode. During working mode the processor speed is set to $\max(\rho_t, s_{\mathrm{cr}})$ until there is no more remaining work. Then speed is set to zero and the processor starts idling. While idling or sleeping, we switch to the working mode only when ρ_t becomes larger than s_{cr}. When the amount of energy spent in the current idle interval equals the transition energy γ (i.e., after time $\gamma/P_{\alpha,\beta}(0)$) the processor is suspended to sleep.

Rejection Policy Let c_1 and c_2 be parameters to be determined later. Consider the arrival of a new job j at time r_j. Reject it immediately if $\delta_j < s_{\mathrm{cr}}^{\alpha-1}/\alpha c_2^{\alpha-1}$. Otherwise, define the *current idle cost* $x \in [0, \gamma]$ depending on the processor's

```
{ at any time $t$ and for $x$ equal to current idle cost }
on arrival of job $j$:
    { let $s_{\mathrm{OA}}$ be $\mathrm{OA}^t$'s speed for $j$ if it were accepted }
    reject if $\delta_j < s_{\mathrm{cr}}^{\alpha-1}/\alpha c_2^{\alpha-1}$ or $v_j < c_1 x$ or $s_{\mathrm{OA}} > c_2 s_{j,\mathrm{p}}$

depending on current mode:
    { let $\rho_t$ denote $\mathrm{OA}^t$'s speed planned for the current time $t$ }
    working:
        if no remaining work:   switch to idle mode
        otherwise:   work at speed $\max(\rho_t, s_{\mathrm{cr}})$ with earliest deadline first
    idling:
        if $x \geq \gamma$ :   switch to sleep mode
        if $\rho_t > s_{\mathrm{cr}}$:   switch to work mode
    sleeping:
        if $\rho_t > s_{\mathrm{cr}}$:   switch to work mode
```

Listing 4.1: Rejection-oblivious online scheduler A.

state as follows: (i) zero if it is working, (ii) the length of the current idle interval times β if it is idle, and (iii) γ if it is asleep. If $v_j < c_1 x$, the job is rejected. Otherwise, compute the job's speed s_{OA} which would be assigned by OA^{r_j} if it were accepted. Reject the job if $s_{\mathrm{OA}} > c_2 s_{j,\mathrm{p}}$, accept otherwise.

4.4.1 Bounding the Different Cost Portions

In the following, let O denote an optimal schedule. Remember that $\mathrm{cost}(A) = E^A_{\mathrm{sleep}} + E^A_{\mathrm{idle}} + E^A_{\mathrm{work}} + V^A_{\mathrm{rej}}$. We bound each of the three terms $E^A_{\mathrm{sleep}} + E^A_{\mathrm{idle}}$, E^A_{work}, and V^A_{rej} separately in Lemma 4.3, Lemma 4.4, and Lemma 4.7, respectively. Eventually, Section 4.4.2 combines these bounds and yields our main result: a nearly tight competitive ratio depending on the maximum value density of the problem instance.

Lemma 4.3 (Sleep and Idle Energy). $E^A_{sleep} + E^A_{idle} \leq 6E^O_{sleep} + 2E^O_{sys} + \frac{4}{c_1} V^O_{rej}$

Proof. Let us first consider E^A_{idle}. Partition the set of idle intervals under schedule A into three disjoint subsets $\mathcal{I}_1$, $\mathcal{I}_2$, and $\mathcal{I}_3$ as follows:

- $\mathcal{I}_1$ contains idle intervals not intersecting any sleep interval of O. By definition, the total length of idle intervals from $\mathcal{I}_1$ is bounded by the time O is awake. Thus, the total cost of $\mathcal{I}_1$ is at most E^O_{sys}.

- For each sleep interval I of O, $\mathcal{I}_2$ contains any idle interval X that is not the last idle interval having a nonempty intersection with I and that is completely contained within I (note that the former requirement is redundant if the last intersecting idle interval is not completely contained in I). Consider any $X \in \mathcal{I}_2$ intersecting I and let j denote the first job processed by A after X. It is easy to see that we must have $[r_j, d_j) \subseteq I$. Thus, O has rejected j. But since A accepted j, we must have $v_j \geq c_1|X|\beta$. This implies that the total cost of $\mathcal{I}_2$ cannot exceed V^O_{rej}/c_1.

- $\mathcal{I}_3$ contains all remaining idle intervals. By definition, the first sleep interval of O can intersect at most one such idle interval, while the remaining sleep intervals of O can be intersected by at most two such idle intervals. Thus, if m denotes the number of sleep intervals under schedule O, we get $|\mathcal{I}_3| \leq 2m - 1$. Our sleeping strategy ensures that the cost of each single idle interval is at most γ. Using this and the definition of sleeping energy, the total cost of $\mathcal{I}_3$ is upper bounded by $(2m-1)\gamma = 2E^O_{\text{sleep}} + \gamma$.

Together, we get $E^A_{\text{idle}} \leq E^O_{\text{sys}} + V^O_{\text{rej}}/c_1 + 2E^O_{\text{sleep}} + \gamma$. Moreover, without loss of generality we can bound γ by $V^O_{\text{rej}}/c_1 + E^O_{\text{sleep}}$: if not both A and O reject all incoming jobs (in which case A would be optimal), O will either accept at least one job and thus wake up ($\gamma \leq E^O_{\text{sleep}}$) or reject the first job A accepts ($\gamma \leq V^O_{\text{rej}}/c_1$). This yields $E^A_{\text{idle}} \leq E^O_{\text{sys}} + 2V^O_{\text{rej}}/c_1 + 3E^O_{\text{sleep}}$. For E^A_{sleep}, note that any but the first of A's sleep intervals is preceded by an idle interval of length $\gamma/P_{\alpha,\beta}(0)$. Each such idle interval has cost γ, so we get $E^A_{\text{sleep}} \leq E^A_{\text{idle}}$. The lemma's statement follows by combining the bounds for E^A_{idle} and E^A_{sleep}. □

Lemma 4.4 (Working Energy). *$E^A_{work} \leq \alpha^\alpha E^O_{work} + c_2^{\alpha-1}\alpha^2 V^O_{rej}$*

The proof of Lemma 4.4 is based on the standard amortized local competitiveness argument, first used by Bansal et al. [BKP07]. Although technically quite similar to the typical argumentation, our proof must carefully consider the more complicated rejection policy (compared to [CLL10a]), while simultaneously handle the different processor states.

Given a schedule S, let $E^S_{\text{work}}(t)$ denote the working energy spent until time t and $V^S_{\text{rej}}(t)$ the discarded value until time t. We show that at any time $t \in \mathbb{R}_{\geq 0}$

the *amortized energy inequality*

$$E^A_{\text{work}}(t) + \varPhi(t) \leq \alpha^\alpha E^O_{\text{work}}(t) + c_2^{\alpha-1}\alpha^2 V^O_{\text{rej}}(t) \tag{4.3}$$

holds. Here, $\varPhi$ is a potential function to be defined in a suitable way. It is constructed such that the following conditions hold:

(i) *Boundary Condition:* At the beginning and end we have $\varPhi(t) = 0$.

(ii) *Running Condition:* At any time t when no job arrives we have

$$\frac{\mathrm{d}E^A_{\text{work}}(t)}{\mathrm{d}t} + \frac{\mathrm{d}\varPhi(t)}{\mathrm{d}t} \leq \alpha^\alpha \frac{\mathrm{d}E^O_{\text{work}}(t)}{\mathrm{d}t} + c_2^{\alpha-1}\alpha^2 \frac{\mathrm{d}V^O_{\text{rej}}(t)}{\mathrm{d}t}. \tag{4.4}$$

(iii) *Arrival Condition:* At any time t when a job arrives we have

$$\varDelta E^A_{\text{work}}(t) + \varDelta\varPhi(t) \leq \alpha^\alpha \varDelta E^O_{\text{work}}(t) + c_2^{\alpha-1}\alpha^2 \varDelta V^O_{\text{rej}}(t). \tag{4.5}$$

The $\varDelta$-terms denote the corresponding change caused by the job arrival.

Once these are proven, amortized energy inequality follows by induction: It obviously holds for $t = 0$, and Conditions (ii) and (iii) ensure that it is never violated. Applying Condition (i) yields Lemma 4.4. The crucial part is to define a suitable potential function $\varPhi$. Our analysis combines aspects from both [CLL10a] and [Han+10]. Different rejection decisions of our algorithm A and the optimal algorithm O require us to handle possibly different job sets in the analysis, while the sleep management calls for a careful handling of the processor's current state.

Construction of $\varPhi$ Consider an arbitrary time $t \in \mathbb{R}_{\geq 0}$. Let $w^A_t(t_1, t_2)$ denote the remaining work at time t accepted by schedule A with deadline in $(t_1, t_2]$. We call the expression $\frac{w^A_t(t_1,t_2)}{t_2-t_1}$ the *density* of the interval $(t_1, t_2]$. Next, we define *critical intervals* $(\tau_{i-1}, \tau_i]$. For this purpose, set $\tau_0 := t$ and define τ_i iteratively to be the maximum time that maximizes the density $\rho_i := \frac{w^A_t(\tau_{i-1},\tau_i)}{\tau_i-\tau_{i-1}}$ of the interval $(\tau_{i-1}, \tau_i]$. We end at the first index l with $\rho_l \leq s_{\text{cr}}$ and set $\tau_l = \infty$ and $\rho_l = s_{\text{cr}}$. Note that $\rho_1 > \rho_2 > \dots > \rho_l = s_{\text{cr}}$. Now, for a schedule S let $w^S_t(i)$ denote the remaining work at time t with deadline in the i-th critical interval $(\tau_{i-1}, \tau_i]$ accepted by schedule S. The potential function is defined as $\Phi(t) := \alpha \sum_{i=1}^{l} \rho_i^{\alpha-1}\big(w_t^A(i) - \alpha w_t^O(i)\big)$. It quantifies how far A is ahead or behind in terms of energy. The densities ρ_i essentially correspond to OA's speed levels, but are adjusted to A's usage of OA. Note that whenever A is in working mode its speed equals $\rho_1 \geq s_{\text{cr}}$.

It remains to prove the boundary, running, and arrival conditions. The boundary condition is trivially true as both A and O have no remaining work at the beginning and end. For the running and arrival conditions, see Propositions 4.5 and 4.6, respectively.

Proposition 4.5. *The running condition holds. That is, at any time t when no job arrives we have*

$$\frac{\mathrm{d}E_{work}^A(t)}{\mathrm{d}t} + \frac{\mathrm{d}\Phi(t)}{\mathrm{d}t} \leq \alpha^\alpha \frac{\mathrm{d}E_{work}^O(t)}{\mathrm{d}t} + c_2^{\alpha-1}\alpha^2 \frac{\mathrm{d}V_{rej}^O(t)}{\mathrm{d}t}. \tag{4.6}$$

Proof. Because no jobs arrive we have $\frac{\mathrm{d}V_{\text{rej}}^O(t)}{\mathrm{d}t} = 0$. Let s_A denote the speed of A and s_O the speed of O. Depending on these speeds, we distinguish four cases:

Case 1: $s_O = 0, s_A = 0$

In this case $\frac{\mathrm{d}E_{\text{work}}^A(t)}{\mathrm{d}t} = \frac{\mathrm{d}E_{\text{work}}^O(t)}{\mathrm{d}t} = \frac{\mathrm{d}\Phi(t)}{\mathrm{d}t} = 0$. Thus, the Running Condition (4.4) holds.

Case 2: $s_O = 0, s_A > 0$

Since $s_A > 0$, algorithm A is in working mode and we have $s_A = \rho_1 \geq s_{\text{cr}}$. Moreover, $\frac{\mathrm{d}E_{\text{work}}^A(t)}{\mathrm{d}t} = P_{\alpha,\beta}(s_A)$, $\frac{\mathrm{d}\Phi(t)}{\mathrm{d}t} = -\alpha s_A^\alpha$, and $\frac{\mathrm{d}E_{\text{work}}^O(t)}{\mathrm{d}t} = 0$. We get

$$\begin{aligned}&\frac{\mathrm{d}E_{\text{work}}^A(t)}{\mathrm{d}t} + \frac{\mathrm{d}\Phi(t)}{\mathrm{d}t} - \alpha^\alpha \frac{\mathrm{d}E_{\text{work}}^O(t)}{\mathrm{d}t} = P_{\alpha,\beta}(s_A) - \alpha s_A^\alpha \\ &= \beta - (\alpha-1)s_A^\alpha \leq \beta - (\alpha-1)s_{\text{cr}}^\alpha = 0.\end{aligned} \tag{4.7}$$

Case 3: $s_O > 0, s_A = 0$

In this case $l = 1$ and thus $\rho_1 = s_{\text{cr}}$. The terms in Inequality (4.4) become $\frac{\mathrm{d}E_{\text{work}}^A(t)}{\mathrm{d}t} = 0$, $\frac{\mathrm{d}\Phi(t)}{\mathrm{d}t} = \alpha^2 s_{\text{cr}}^{\alpha-1} s_O$, and $\frac{\mathrm{d}E_{\text{work}}^O(t)}{\mathrm{d}t} = P_{\alpha,\beta}(s_O)$. We get

$$\begin{aligned}&\frac{\mathrm{d}E_{\text{work}}^A(t)}{\mathrm{d}t} + \frac{\mathrm{d}\Phi(t)}{\mathrm{d}t} - \alpha^\alpha \frac{\mathrm{d}E_{\text{work}}^O(t)}{\mathrm{d}t} = \alpha^2 s_{\text{cr}}^{\alpha-1} s_O - \alpha^\alpha P_{\alpha,\beta}(s_O) \\ &\leq \alpha^2 s_{\text{cr}}^{\alpha-1} s_O - \alpha^\alpha s_O \frac{P_{\alpha,\beta}(s_{\text{cr}})}{s_{\text{cr}}} \leq s_O s_{\text{cr}}^{\alpha-1}(\alpha^2 - \alpha^\alpha) \leq 0.\end{aligned} \tag{4.8}$$

Case 4: $s_O > 0, s_A > 0$

Because of $s_A > 0$ we know A is in the working state and, thus, $s_A = \rho_1 \geq s_{\text{cr}}$. So, this time we have $\frac{\mathrm{d}E^A_{\text{work}}(t)}{\mathrm{d}t} = P_{\alpha,\beta}(s_A)$, $\frac{\mathrm{d}\Phi(t)}{\mathrm{d}t} = -\alpha s_A^\alpha + \alpha^2 \rho_k^{\alpha-1} s_O$, and $\frac{\mathrm{d}E^O_{\text{work}}(t)}{\mathrm{d}t} = P_{\alpha,\beta}(s_O)$. We get

$$\begin{aligned} &\frac{\mathrm{d}E^A_{\text{work}}(t)}{\mathrm{d}t} + \frac{\mathrm{d}\Phi(t)}{\mathrm{d}t} - \alpha^\alpha \frac{\mathrm{d}E^O_{\text{work}}(t)}{\mathrm{d}t} \\ &= P_{\alpha,\beta}(s_A) - \alpha s_A^\alpha + \alpha^2 \rho_k^{\alpha-1} s_O - \alpha^\alpha P_{\alpha,\beta}(s_O) \\ &\leq s_A^\alpha - \alpha s_A^\alpha + \alpha^2 s_A^{\alpha-1} s_O - \alpha^\alpha s_O^\alpha \leq 0. \end{aligned} \tag{4.9}$$

The last inequality follows from the same argument as in [BKP07]: Divide by s_O^α and substitute $z = {}^{s_A}/_{s_O}$. It becomes equivalent to $(1-\alpha)z^\alpha + \alpha^2 z^{\alpha-1} - \alpha^\alpha \leq 0$. Differentiating with respect to z yields the correctness. □

Proposition 4.6. *The arrival condition holds. That is, at any time t when a job arrives we have*

$$\Delta E^A_{work}(t) + \Delta\Phi(t) \leq \alpha^\alpha \Delta E^O_{work}(t) + c_2^{\alpha-1}\alpha^2 \Delta V^O_{rej}(t). \tag{4.10}$$

Here, the Δ-terms denote the corresponding change caused by the job arrival.

Proof. The arrival of a job j does not change the energy invested so far, thus $\Delta E^A_{\text{work}}(t) = 0$ and $\Delta E^O_{\text{work}}(t) = 0$. If A rejects j, we have $\Delta\Phi(t) \leq 0$ and $\Delta V^O_{\text{rej}}(t) \geq 0$, thus the Arrival Condition (4.5) holds. So assume A accepts j. The arrival of j may change the critical intervals and their densities significantly. However, as pointed out in [BKP07], these changes can be broken down into a series of simpler changes affecting at most two adjacent critical intervals. Thus, we first consider the effect of arrivals which do not change the critical intervals. Afterward, we use the technique from [BKP07] to reduce an arbitrary change to these simple cases.

Case 1: The critical intervals remain unchanged and only ρ_k for $k < l$ changes. Let ρ_k and ρ'_k denote the densities of $(\tau_{k-1}, \tau_k]$ just before and after j's arrival, respectively. That is, $\rho'_k = \rho_k + \frac{w_j}{\tau_k - \tau_{k-1}}$. Note that ρ'_k is the speed planned by OA^t for job j. Because A accepted j we have $\rho'_k \leq c_2 s_{j,\text{p}}$. If j is rejected by O, we have $\Delta V^O_{\text{rej}}(t) = v_j$. Since only the k-th critical interval is affected, the change in the potential function is given by

$$\Delta\Phi(t) = \alpha\rho_k'^{\alpha-1}\big(w_t^A(k) + w_j - \alpha w_t^O(k)\big) - \alpha\rho_k^{\alpha-1}\big(w_t^A(k) - \alpha w_t^O(k)\big).$$

Now, we compute, analogously to Lemma 4 in [CLL10a], that $\Delta\Phi(t)$ equals

$$\begin{aligned}
&\alpha\rho_k'^{\alpha-1}\big(w_t^A(k) + w_j - \alpha w_t^O(k)\big) - \alpha\rho_k^{\alpha-1}\big(w_t^A(k) - \alpha w_t^O(k)\big)\\
&\le \alpha\rho_k'^{\alpha-1}\big(w_t^A(k) + w_j\big) - \alpha\rho_k^{\alpha-1}w_t^A(k) = \alpha\frac{\big(w_t^A(k)+w_j\big)^\alpha - w_t^A(k)^\alpha}{(\tau_k-\tau_{k-1})^{\alpha-1}}\\
&\le \alpha^2\frac{\big(w_t^A(k)+w_j\big)^{\alpha-1}w_j}{(\tau_k-\tau_{k-1})^{\alpha-1}} = \alpha^2\rho_k'^{\alpha-1}w_j \le \alpha^2(c_2 s_{j,\mathrm{p}})^{\alpha-1}w_j = c_2^{\alpha-1}\alpha^2 v_j.
\end{aligned}$$

The penultimate inequality uses the fact that $f(x) = x^\alpha$ is convex, yielding $f(y) - f(x) \le f'(y)\cdot(y-x)$ for all $y > x$. This implies the Arrival Condition (4.5). If j is accepted by O, we have $\Delta V_{\mathrm{rej}}^O(t) = 0$ and $\Delta\Phi(t)$ becomes

$$\alpha\rho_k'^{\alpha-1}\Big(w_t^A(k) + w_j - \alpha\big(w_t^O(k) + w_j\big)\Big) - \alpha\rho_k^{\alpha-1}\big(w_t^A(k) - \alpha w_t^O(k)\big).$$

In the same way as in [BKP07], we now get $\Delta\Phi(t) \le 0$.

Case 2: Only the amount of work assigned to the last critical interval increases.

Remember that, by definition, $\tau_l = \infty$ and $\rho_l = s_{\mathrm{cr}}$. Since j is accepted by A we have $s_{\mathrm{cr}}^{\alpha-1} \le \alpha c_2^{\alpha-1}\delta_j$. If O rejects j we have $\Delta V_{\mathrm{rej}}^O(t) = v_j$ and the Arrival Condition (4.5) follows from

$$\begin{aligned}
\Delta\Phi(t) &= \alpha\rho_l^{\alpha-1}\big(w_t^A(l) + w_j - \alpha w_t^O(l)\big) - \alpha\rho_l^{\alpha-1}\big(w_t^A(l) - \alpha w_t^O(l)\big)\\
&= \alpha s_{\mathrm{cr}}^{\alpha-1}w_j \le \alpha^2 c_2^{\alpha-1}\delta_j w_j = c_2^{\alpha-1}\alpha^2 v_j.
\end{aligned}$$

If O accepts j, $\Delta V_{\mathrm{rej}}^O(t) = 0$ and the Arrival Condition (4.5) is implied by

$$\begin{aligned}
\Delta\Phi(t) &= \alpha\rho_l^{\alpha-1}\Big(w_t^A(l) + w_j - \alpha\big(w_t^O(l) + w_j\big)\Big) - \alpha\rho_l^{\alpha-1}\big(w_t^A(l) - \alpha w_t^O(l)\big)\\
&= \alpha\rho_l^{\alpha-1}w_j(1-\alpha) \le 0.
\end{aligned}$$

Let us now consider the arrival of an arbitrary job j. The idea is to split this job into two jobs j_1 and j_2 with the same release time, deadline, and value density as j. Their total work equals w_j. Let x denote the size of j_1. We determine a suitable x by continuously increasing x from 0 to w_j until two critical intervals merge or one critical interval splits. The arrival of j_1 can then be handled by one of the above cases, while j_2 is treated recursively in the same way as j. For details, see [BKP07] or [Han+10]. $\square$

Bounding the Rejected Value In the following we bound the total value V^A_{rej} of jobs rejected by A. The general idea is similar to the one by Chan et al. [CLL10a]. However, in contrast to the simpler model without sleep states, we must handle small-valued jobs of high density explicitly (cf. Section 4.3). Moreover, the sleeping policy introduces an additional difficulty: our algorithm does not preserve all structural properties of an OA schedule (cf. Fact 4.1). This prohibits a direct mapping between the energy consumption of algorithm A and of the intermediate OA schedules during a *fixed time interval*, as used in the corresponding proof in [CLL10a]. Indeed, the actual energy used by A during a fixed time interval may decrease compared to the energy planned by the intermediate OA schedule, as A may decide to raise the speed to s_{cr} at certain points in the schedule. Thus, to bound the value of a job rejected by A but processed by the optimal algorithm for a relatively long time, we have to consider the energy usage for the workload OA planned for that time (instead of the actual energy usage for the workload A processed during that time, which might be quite different).

Lemma 4.7 (Rejected Value)**.** *Let δ_{max} be the maximum value density of jobs of value less than $c_1\gamma$ and consider an arbitrary parameter $b \geq {}^1/_{c_2}$. Then A's rejected value is at most*

$$V^A_{rej} \leq \max\left(\delta_{max}\frac{s_{cr}}{P_{\alpha,\beta}(s_{cr})}, b^{\alpha-1}\right)E^O_{work} + \frac{b^{\alpha-1}}{(c_2 b-1)^\alpha}E^A_{work} + V^O_{rej}. \tag{4.11}$$

Proof. Partition the jobs rejected by A into two disjoint subsets J_1 (jobs rejected by both A and O) and J_2 (jobs rejected by A only). The total value of jobs in J_1 is at most V^O_{rej}. Thus, it suffices to show that the total value of J_2 is bounded by

$$\max\left(\delta_{\max}\tfrac{s_{\text{cr}}}{P_{\alpha,\beta}(s_{\text{cr}})}, b^{\alpha-1}\right)E^O_{\text{work}} + \tfrac{b^{\alpha-1}}{(c_2 b-1)^\alpha}E^A_{\text{work}}. \tag{4.12}$$

To this end, let $j \in J_2$. Remember that, because of the convexity of the power function, O can be assumed to process j at a constant speed s_O. Otherwise processing j at its average speed could only improve the schedule. Let us distinguish three cases, depending on the reason for which A rejected j:

Case 1: j got rejected because of $\delta_j < \frac{s_{\mathrm{cr}}^{\alpha-1}}{\alpha c_2^{\alpha-1}}$.

Let $E_{\mathrm{work}}^{O}(j)$ denote the working energy invested by O into job j. Using the rejection condition we can compute

$$E_{\mathrm{work}}^{O}(j) = \frac{P_{\alpha,\beta}(s_O)}{s_O} w_j \geq \frac{P_{\alpha,\beta}(s_{\mathrm{cr}})}{s_{\mathrm{cr}}} w_j \geq s_{\mathrm{cr}}^{\alpha-1} w_j > \alpha c_2^{\alpha-1} v_j. \tag{4.13}$$

Together with $b \geq {}^{1}/_{c_2}$ we get $v_j < b^{\alpha-1} E_{\mathrm{work}}^{O}(j)$.

Case 2: j got rejected because of $v_j < c_1 x$

As in the algorithm description, let $x \in [0, \gamma]$ denote the current idle cost at time r_j. Since j's value is less than $c_1 x \leq c_1 \gamma$, we have $\delta_j \leq \delta_{\max}$. We get

$$E_{\mathrm{work}}^{O}(j) = \frac{P_{\alpha,\beta}(s_O)}{s_O} w_j = \frac{P_{\alpha,\beta}(s_O)}{s_O} \frac{v_j}{\delta_j} \geq \frac{P_{\alpha,\beta}(s_{\mathrm{cr}})}{s_{\mathrm{cr}}} \frac{v_j}{\delta_{\max}}, \tag{4.14}$$

which eventually yields $v_j \leq \delta_{\max} \frac{s_{\mathrm{cr}}}{P_{\alpha,\beta}(s_{\mathrm{cr}})} E_{\mathrm{work}}^{O}(j)$.

Case 3: j got rejected because of $s_{\mathrm{OA}} > c_2 s_{j,\mathrm{p}}$

Here, s_{OA} denotes the speed OA^{r_j} would assign to j if it were accepted. We use $\mathrm{OA}^{\underline{r_j}}$ to refer to the OA schedule at time r_j without j. Let $b_j := {}^{s_{j,\mathrm{p}}}/_{s_O}$. We bound v_j in different ways, depending on b_j. If b_j is small (i.e., $b_j \leq b$) we use

$$E_{\mathrm{work}}^{O}(j) \geq \frac{P_{\alpha,0}(s_O)}{s_O} w_j = \frac{P_{\alpha,0}({}^{s_{j,\mathrm{p}}}/_{b_j})}{{}^{s_{j,\mathrm{p}}}/_{b_j}} w_j = \frac{s_{j,\mathrm{p}}^{\alpha-1}}{b_j^{\alpha-1}} w_j = \frac{v_j}{b_j^{\alpha-1}}. \tag{4.15}$$

That is, we have $v_j \leq b_j^{\alpha-1} E_{\mathrm{work}}^{O}(j)$. Otherwise, if b_j is relatively large, v_j is bounded by E_{work}^{A}. Let I denote the period of time when O processes j at constant speed s_O and let W denote the work processed by $\mathrm{OA}^{\underline{r_j}}$ during this time. Since $I \subseteq [r_j, d_j)$, Fact 4.1(b) implies that OA^{r_j}'s speed during I is at least $s_{\mathrm{OA}} > c_2 s_{j,\mathrm{p}}$. Thus, the total amount of work processed by OA^{r_j} during I is larger than $c_2 s_{j,\mathrm{p}} |I|$. But then, by applying Fact 4.1(c), we see that W must be larger than $c_2 s_{j,\mathrm{p}} |I| - w_j$. Now, W is a subset of the work processed by A. Moreover, Fact 4.1(d) and the definition of algorithm A

ensure that the speeds used for this work in schedule A cannot be smaller than the ones used in $\mathrm{OA}^{r_j}_{\underline{j}}$. In particular, the average speed $s_\emptyset$ used for this work in schedule A is at least $W/|I|$ (the average speed used by $\mathrm{OA}^{r_j}_{\underline{j}}$ for this work). Let $E^A_{\text{work}}(W)$ denote the energy invested by schedule A into the work W. Then, by exploiting the convexity of the power function, we get

$$\begin{aligned} E^A_{\text{work}}(W) &\geq \frac{P_{\alpha,\beta}(s_\emptyset)}{s_\emptyset}W \geq \frac{P_{\alpha,0}(s_\emptyset)}{s_\emptyset}W = {s_\emptyset}^{\alpha-1}W \geq \frac{W^{\alpha-1}}{|I|^{\alpha-1}}W = |I|\frac{W^\alpha}{|I|^\alpha} \\ &> |I|(c_2 s_{j,\mathrm{p}} - s_O)^\alpha = \frac{w_j}{s_O}s_O^\alpha(c_2 b_j - 1)^\alpha = \frac{(c_2 b_j - 1)^\alpha}{b_j^{\alpha-1}}v_j. \end{aligned}$$

That is, we have $v_j < \frac{b_j^{\alpha-1}}{(c_2 b_j-1)^\alpha}E^A_{\text{work}}(W)$. Now, let us specify how to choose from these two bounds:

- If $b_j \leq b$, we apply the first bound: $v_j = b_j^{\alpha-1}E^O_{\text{work}}(j) \leq b^{\alpha-1}E^O_{\text{work}}(j)$.
- Otherwise we have $b_j > b \geq 1/c_2$. Note that for $x > 1/c$ the function $f(x) = \frac{x^{\alpha-1}}{(cx-1)^\alpha}$ decreases. Thus, we get $v_j < \frac{b^{\alpha-1}}{(c_2 b-1)^\alpha}E^A_{\text{work}}(W)$.

By combining these cases we get

$$v_j \leq \max\left(\delta_{\max}\frac{s_{\mathrm{cr}}}{P_{\alpha,\beta}(s_{\mathrm{cr}})}, b^{\alpha-1}\right)E^O_{\text{work}}(j) + \frac{b^{\alpha-1}}{(c_2 b-1)^\alpha}E^A_{\text{work}}(W). \tag{4.16}$$

Note that both energies referred to, $E^O_{\text{work}}(j)$ as well as $E^A_{\text{work}}(W)$, are mutually different for different jobs j. Thus, we can combine these inequalities for all jobs $j \in J_2$ to get the desired result. □

4.4.2 Putting it All Together.

The following theorem combines the results of Lemma 4.3, Lemma 4.4, and Lemma 4.7.

Theorem 4.8. *Let $\alpha \geq 2$ and let δ_{max} be the maximum value density of jobs of value less than $c_1\gamma$. Moreover, define $\eta := \max(\delta_{max}\frac{s_{cr}}{P_{\alpha,\beta}(s_{cr})}, b^{\alpha-1})$ and $\mu := \frac{b^{\alpha-1}}{(c_2 b-1)^\alpha}$ for a parameter $b \geq 1/c_2$. Then A's competitive ratio is at most*

$$\max\left(c_2^{\alpha-1}\alpha^2, \alpha^\alpha\right)(1+\mu) + \max(2+\eta, 1+4/c_1). \tag{4.17}$$

Proof. Lemma 4.3 together with the relation $E^O_{\text{sys}} \leq E^O_{\text{idle}} + E^O_{\text{work}}$ bounds the sleep and idle energy of A with respect to O's cost as $E^A_{\text{sleep}} + E^A_{\text{idle}} \leq 6E^O_{\text{sleep}} + 2E^O_{\text{idle}} + 2E^O_{\text{work}} + \frac{4}{c_1}V^O_{\text{rej}}$. For the working energy, Lemma 4.4 yields $E^A_{\text{work}} \leq \alpha^\alpha E^O_{\text{work}} + c_2^{\alpha-1}\alpha^2 V^O_{\text{rej}}$. To bound the total value rejected by A with respect to the cost of O, we apply Lemma 4.4 to Lemma 4.7 and get

$$V^A_{\text{rej}} \leq \eta E^O_{\text{work}} + \mu E^A_{\text{work}} + V^O_{\text{rej}} \leq (\eta + \alpha^\alpha \mu)E^O_{\text{work}} + \left(c_2^{\alpha-1}\alpha^2\mu + 1\right)V^O_{\text{rej}}.$$

Using these inequalities, we can bound the cost of A as follows:

$$\begin{aligned} \operatorname{cost}(A) \leq 6E^O_{\text{sleep}} + 2E^O_{\text{idle}} + (\alpha^\alpha + \alpha^\alpha\mu + 2 + \eta)E^O_{\text{work}} \\ + \left(c_2^{\alpha-1}\alpha^2 + c_2^{\alpha-1}\alpha^2\mu + 1 + {}^4\!/_{c_1}\right)V^O_{\text{rej}}. \end{aligned} \tag{4.18}$$

Since $6 \leq \alpha^\alpha + 2$ for $\alpha \geq 2$, we get the following bound on A's competitive ratio:

$$\frac{\operatorname{cost}(A)}{\operatorname{cost}(O)} \leq \max\left(c_2^{\alpha-1}\alpha^2, \alpha^\alpha\right)(1+\mu) + \max(2+\eta, 1 + {}^4\!/_{c_1}). \tag{4.19}$$

□

By a careful choice of parameters we get a constant competitive ratio if restricting the value density of small-valued jobs accordingly. So, let $\alpha \geq 2$ and set $c_2 = \alpha^{\frac{\alpha-2}{\alpha-1}}$, $b = \frac{\alpha+1}{c_2}$, and $c_1 = \frac{4}{1+b^{\alpha-1}} \leq 1$. Applying Theorem 4.8 using these parameters yields the following results:

Corollary 4.9. *Algorithm A is* $\alpha^\alpha + 2e\alpha + \delta_{max}\frac{s_{cr}}{P_{\alpha,\beta}(s_{cr})}$*-competitive.*

Corollary 4.10. *Algorithm A is* $\alpha^\alpha + 2e\alpha$*-competitive if we restrict it to instances of maximum value density* $\delta_{max} := b^{\alpha-1}\frac{P_{\alpha,\beta}(s_{cr})}{s_{cr}}$*. This competitive ratio still holds if the restriction is only applied to jobs of value less than* $\frac{4}{1+b^{\alpha-1}}\gamma$*.*

Proof. First note the identity $b^{\alpha-1} = \alpha(1 + {}^1\!/_\alpha)^{\alpha-1}$. Moreover, using the definitions from Theorem 4.8, we see that $\eta = b^{\alpha-1}$ and $\alpha^\alpha\mu = b^{\alpha-1}$. By applying Theorem 4.8 to our choice of parameters, the competitive ratio of A becomes

$$\begin{aligned} \alpha^\alpha(1+\mu) + 2 + \eta = \alpha^\alpha + 2 + 2b^{\alpha-1} = \alpha^\alpha + 2\left(1 + \alpha(1 + {}^1\!/_\alpha)^{\alpha-1}\right) \\ \leq \alpha^\alpha + 2\alpha(1 + {}^1\!/_\alpha)^\alpha \leq \alpha^\alpha + 2e\alpha. \end{aligned} \tag{4.20}$$

□

Corollary 4.11. *If only considering instances for which the job values are at least $\frac{8}{2+3\alpha}\gamma \leq \gamma$, A's competitive ratio is at most $\alpha^\alpha + 2e\alpha$.*

Proof. Follows from Corollary 4.10 by using that for $\alpha \geq 2$ we have

$$\frac{4}{1+b^{\alpha-1}} = \frac{4}{1+\alpha(1+{}^1\!/\!_\alpha)^{\alpha-1}} \leq \frac{4}{1+\frac{3}{2}\alpha} = \frac{8}{2+3\alpha}. \qquad \square$$

Note that the bound from Corollary 4.9 is nearly tight with respect to $\delta_{\max}$ and the lower bound from Theorem 4.2.

4.5 The Speed-Bounded Case

As stated earlier, our model can be considered as a generalization of [CLL10a]. It adds sleep states, leading to several structural difficulties which we solved in the previous section. A further, natural generalization of the model is to cap the speed at some maximum speed T. Algorithms based on OA often lend themselves to such bounded speed models. In many cases, a canonical adaptation – possibly mixed with a more involved job selection rule – leads to an algorithm for the speed bounded case with similar properties (see, e.g., [CLL10a; Han+10; Cha+07; Ban+08b]). A notable property of the profit-oriented scheduling model of [CLL10a] is that limiting the maximum speed leads to a non-constant competitive ratio. Instead, it becomes highly dependent on a job's penalty ratio defined as $\Gamma_j := {}^{s_{j,\mathrm{P}}}\!/\!_T$. They derive a lower bound of $\Omega\big(\max({}^{e^{\alpha-1}}\!/\!_\alpha, \Gamma^{\alpha-2+{}^1\!/\!_\alpha})\big)$ where $\Gamma = \max \Gamma_j$. Since our model generalizes their model, this bound transfers immediately to our setting (for the case $\beta = \gamma = 0$). On the positive side we can adapt our algorithm, similar to [CLL10a], by additionally rejecting a job if its speed planned by OA is larger than T (cf. rejection condition in algorithm description, Section 4.4). Our main theorem from Section 4.4 becomes

Theorem 4.12. *Let $\alpha \geq 2$ and let δ_{max} be the maximum value density of jobs of value less than $c_1\gamma$. Moreover, define $\eta := \max(\delta_{max}\frac{s_{cr}}{P_{\alpha,\beta}(s_{cr})}, \Gamma^{\alpha-1}b^{\alpha-1})$ and $\mu := \Gamma^{\alpha-1}\frac{b^{\alpha-1}}{(b-1)^\alpha}$ for $b \geq 1$. Then A's competitive ratio is at most*

$$\alpha^\alpha(1+\mu) + \max(2+\eta, 1+{}^4\!/\!_{c_1}). \tag{4.21}$$

Proof Sketch. Note that the results from Lemmas 4.3 and 4.4 remain valid without any changes, as an additional rejection rule does not influence the corresponding proofs. The only lemma affected by the changed algorithm is Lemma 4.7. In its proof, we have to consider an additional rejection case, namely that job j got rejected because of $s_{\text{OA}} > T = \frac{1}{\Gamma_j} s_{j,\text{p}}$. This can be handled completely analogously to Case 3 in the proof, using the factor $\frac{1}{\Gamma_j}$ instead of c_2. We get the bounds $v_j \leq b_j^{\alpha-1} E^O_{\text{work}}(j)$ and $v_j < b_j^{\alpha-1}/(b_j/\Gamma_j - 1)^{\alpha} E^A_{\text{work}}(W)$. If $b_j \leq \Gamma_j b$ this yields $v_j \leq \Gamma_j^{\alpha-1} b^{\alpha-1} E^O_{\text{work}}(j)$. Otherwise, if $b_j > \Gamma_j b$, we have $v_j < \Gamma_j^{\alpha-1} \frac{b^{\alpha-1}}{(b-1)^{\alpha}} E^A_{\text{work}}(W)$. The remaining argumentation is the same as in the proof of Theorem 4.8. □

For $b = \alpha + 1$ and the interesting case $\Gamma > 1$ we get a competitive ratio of $\alpha^{\alpha}(1 + 2\Gamma^{\alpha-1}) + \delta_{\max} \frac{s_{\text{cr}}}{P_{\alpha,\beta}(s_{\text{cr}})}$. For job values of at most γ it is $\alpha^{\alpha}(1 + 2\Gamma^{\alpha-1})$.

4.6 Conclusion & Outlook

This chapter proposed a new model that combines two modern energy conservation techniques with a profitability aspect. The results point out an inherent connection between the necessary and sufficient competitive ratio of rejection-oblivious algorithms and the maximum value density. A natural question is how far this connection applies to other, more involved algorithm classes. Can we find better strategies if allowed to reject jobs even after we invested some energy, or if taking former rejection decisions into account? Such more involved rejection policies have proven useful in other models [PS10; Han+10], and we conjecture that they would do so in our setting. Other interesting directions include models for multiple processors (cf. Chapter 3) as well as more general power functions. Also, Pruhs and Stein [PS10] modeled job values and deadlines in a more general way, which seems especially interesting for the presented profit-oriented model.

Apart from such model variants, I conjecture that the presented results can be improved. Especially considering the results from Chapter 3, it seems likely that the competitive ratios given in this chapter (especially Corollary 4.11 and its pendants) can be improved to $\alpha^{\alpha} + 2$.

CHAPTER 5

Trading Energy for Responsiveness

> *There's ways to amuse yourself while doing things and that's how I look at efficiency.*
>
> Donald Ervin Knuth

EFFICIENCY occurs in various forms, and it depends on the task at hand which to consider. Typically it involves some kind of resource and a measure to evaluate a solution's quality. The resource we consider is essentially fixed by the thesis title: Energy[1]. With respect to the quality measure, previous chapters considered deadline scheduling together with some kind of profit. The present chapter changes focus and studies a quality measure that incorporates a trade-off between energy and responsiveness. More precisely, we will consider a linear combination of the total energy consumption and the average response time of all jobs. Intuitively, this linear combination fixes the amount of energy we are willing to trade for improving the average response time by one time unit.

Model Differences & Fractional Flow The model and problem of this chapter, which we formally introduce in Section 5.2, differs slightly from what

[1] Although, we will slightly deviate from this route in Chapter 6.

we considered previously. To start with, while jobs still come with release times, we are no longer concerned with the online case but with the offline setting. That is, the total number of jobs and all their properties are known a priori. Moreover, instead of the continuous speed scaling model of Chapters 3 and 4, we consider a (single) processor with a discrete number of possible speed levels, each with its own constant power consumption. Note that this is actually a much more realistic model than the continuous one, as most speed-scalable real-world processors provide a fixed set of discrete speed levels. We will discuss this issue more thoroughly together with the related work in Section 5.1.

The objective of average or, equivalently, total response time is also known as total flow time or simply *total flow*. Actually, our objective will consider fractional flow. Intuitively, one can think of it as the average response time over all (infinitesimal small) units of work.

Chapter Basis The model, analysis, and results presented in the remainder of this chapter are based on the following publication:

> **2014** (with A. Antoniadis, N. Barcelo, M. Consuegra, M. Nugent, K. Pruhs and M. Scquizzato). "Efficient Computation of Optimal Energy and Fractional Weighted Flow Trade-off Schedules". In: *Proceedings of the 31st Symposium on Theoretical Aspects of Computer Science (STACS)*. in press, cf. [Ant+14].

Chapter Outline Section 5.1 explores further literature and its relation to the results of this chapter. A formal model description can be found in Section 5.2. The remaining sections deal with the description and analysis of our main result, a polynomial-time algorithm for our discrete speed scaling problem. While the presented algorithm will turn out to have a very clean geometric interpretation, its design and analysis are rather complicated. Thus, we start with an overview of the main conceptual ideas in Section 5.3 before launching into details in the subsequent sections. In Section 5.4, we present the obvious linear programming formulation of the problem, and discuss our interpretation of information that can be gained about optimal schedules from both the primal and dual linear programs. This information is used in Section 5.5 to develop a polynomial-time algorithm that computes an optimal solution. Section 5.6 is devoted to the correctness and Section 5.7 to the running time of our algorithm. Finally, Section 5.8 gives a short résumé of this chapter and the presented results.

5.1 Related Work & Contribution

We continue with an overview of related work and how it relates to the results of this chapter, especially with respect to speed scaling and flow time objectives. Once more, we keep the overview self-contained. See Chapter 2 for a more general literature overview.

There seem to be essentially three papers in the algorithmic literature that study the computation of optimal energy trade-off schedules. All assume that the processor can run at any non-negative real speed, and that the power used by the processor is some nice function of the speed (most commonly the speed raised to some constant α). Both Albers and Fujiwara [AF07] as well as Pruhs et al. [PUW08] give polynomial-time algorithms for the special case of our problem where the densities of all units of work are the same. Pruhs et al. [PUW08] present a homotopic optimization algorithm that, intuitively, traces out all schedules that are Pareto-optimal with respect to energy and fractional flow, one of which must obviously be the optimal energy trade-off schedule. In [AF07], the authors consider a dynamic programming algorithm and introduce the notion of trade-off schedules. The third paper to be mentioned is due to Barcelo et al. [Bar+13] and gives a polynomial-time algorithm for recognizing an optimal schedule. The authors also show that the optimal schedule evolves continuously as a function of the importance of energy, implying that a continuous homotopic algorithm is, at least in principle, possible. However, [Bar+13] could not provide any bound, even exponential, on the time of this algorithm, nor were the authors able to provide any way to discretize this algorithm.

To reemphasize, the prior literature [AF07; PUW08; Bar+13] assumes that the set of allowable speeds is continuous. Our setting of discrete speeds models the current technology more closely and appears to be algorithmically more challenging. In [Bar+13] the recognition of an optimal trade-off schedule in the continuous setting is a direct consequence of the KKT conditions of the natural convex program, as it is observed that there is essentially only one degree of freedom for each job in any plausibly optimal schedule. This degree of freedom can be recovered from the candidate schedule by looking at the job's speed function (describing when and at what speed it is run). In the discrete setting, we shall see that there is again essentially only one degree of freedom for each job, but unfortunately one cannot easily recover the value of this degree of freedom by examining the candidate schedule. Thus, we do not know of any simple way to even recognize an optimal trade-off schedule in the discrete setting.

One might also reasonably consider the performance measure of the aggregate weighted flow over jobs (instead of work), where the flow of a job is the amount of time between when the job is released and when the last bit of work of that job is finished. In the context that the jobs are flight queries to a travel site, aggregating over the delay of jobs is probably more appropriate in the case of Orbitz, as Orbitz does not present the querier with any information until all the possible flights are available, while aggregating over the delay of work may be more appropriate in the case of Kayak, as Kayak presents the querier with flight options as they are found. Also, often the aggregate flow of work is used as a surrogate measure for the aggregate flow of jobs as it tends to be more mathematically tractable. In particular, for the trade-off problem that we consider here, the problem is NP-hard if we were to consider the performance measure of the aggregate weighted flow of jobs, instead of the aggregate weighted flow of work. The hardness follows immediately from the well-known fact that minimizing the weighted flow time of jobs on a unit speed processor is NP-hard [Lab+84], or from the fact that minimizing total weighted flow (without release times) subject to an energy budget is NP-hard [MV13].

There is a fair number of papers that study approximately computing optimal trade-off schedules, both offline and online. For example, Megow and Verschae [MV13] give polynomial-time approximation schemes for minimizing total flow without release times subject to an energy budget in both the continuous and discrete speed settings. For the online setting, literature can be distinguished by whether they consider total flow and energy [AF07; BPS09; Lam+08; Ban+08b; AWT09; BCP13; CLL10b; DH14] or fractional flow and energy [Ban+08b; BCP13]. More details about energy-efficient algorithms can by found in the survey by Albers [Alb10].

5.2 Model & Preliminaries

We consider the problem of scheduling a set $\mathcal{J} := \{1, 2, \dots, n\}$ of n jobs on a single processor featuring k different speeds $0 < s_1 < s_2 < \dots < s_k$. The power consumption of the processor while running at speed s_i is $P_i \geq 0$. We use $\mathcal{S} := \{s_1, \dots, s_k\}$ to denote the set of speeds and $\mathcal{P} := \{P_1, \dots, P_k\}$ to denote the set of powers. While running at speed s_i, the processor performs s_i units of work per time unit and consumes energy at a rate of P_i.

Each job $j \in \mathcal{J}$ has a release time r_j, a processing volume (or work) p_j, and a weight w_j. Moreover, we denote the value $d_j := {}^{w_j}/_{p_j}$ as the density of job j. All densities are, without loss of generality, distinct[2]. For each time t, a schedule S must decide which job to process at what speed. Preemption is allowed, so that a job may be suspended at any point in time and resumed later on. We model a schedule S by a speed function $S\colon \mathbb{R}_{\geq 0} \to \mathcal{S}$ and a scheduling policy $J\colon \mathbb{R}_{\geq 0} \to \mathcal{J}$. Here, $S(t)$ denotes the speed at time t and $J(t)$ the job that is scheduled at time t. Jobs can be processed only after they have been released. For job j let $I_j := J^{-1}(j) \cap [r_j, \infty)$ be the union of time intervals during which it is processed. A feasible schedule must finish the work of all jobs. That is, the inequality $\int_{I_j} S(t)\,\mathrm{d}t \geq p_j$ must hold for all jobs j.

We measure the quality of a given schedule S by means of its energy consumption and its fractional flow. The speed function $S\colon \mathbb{R}_{\geq 0} \to \mathcal{S}$ induces a power function $P\colon \mathbb{R}_{\geq 0} \to \mathcal{P}$, such that $P(t)$ is the power consumed at time t. The energy consumption of schedule S is $E(S) := \int_0^\infty P(t)\,\mathrm{d}t$. The flow time (also called response time) of a job j is the difference between its completion time and release time. If F_j denotes the flow time of job j, the weighted flow of schedule S is $\sum_{j \in \mathcal{J}} w_j F_j$. However, we are interested in the fractional flow, which takes into account that different parts of a job j finish at different times. More formally, if $p_j(t)$ denotes the work of job j that is processed at time t (i.e., $p_j(t) = S(t)$ if $J(t) = j$, and $p_j(t) = 0$ otherwise), the fractional flow time of job j is $\tilde{F}_j := \int_{r_j}^\infty (t - r_j) \frac{p_j(t)}{p_j}\,\mathrm{d}t$. The fractional weighted flow of schedule S is $\tilde{F}(S) := \sum_{j \in \mathcal{J}} w_j \tilde{F}_j$. The objective function is $E(S) + \tilde{F}(S)$. Our goal is to find a feasible schedule that minimizes this objective.

We define $s_0 := 0$, $P_0 := 0$, $s_{k+1} := s_k$, and $P_{k+1} := \infty$ to simplify notation.

[2]It is not hard to adapt the algorithm if equal densities might occur; see the full version of [Ant+14] for further details.

Note that, without loss of generality, we can assume $\frac{P_i - P_{i-1}}{s_i - s_{i-1}} < \frac{P_{i+1} - P_i}{s_{i+1} - s_i}$. Otherwise, any schedule using s_i could be improved by linearly interpolating the speeds s_{i-1} and s_{i+1}. Moreover, we also assume that jobs are ordered by decreasing density (i.e., $d_1 > d_2 > \cdots > d_n$).

5.3 Overview

The proposed algorithm turns out to have a very nice and simple geometrical interpretation, but also to be quite technical to write down and to analyze. Thus, this section provides an overview of our general approach and its intuition. The reader is advised to start with this overview to get a first idea of what to anticipate in the remaining sections. The overview is roughly structured in resemblance of the remaining chapter, to make it easier to come back and regain a clear picture of the analysis' state.

Geometric Interpretation Our algorithm is based on a natural linear programming formulation of the problem. By considering the corresponding dual program and its complementary slackness conditions, we find necessary and sufficient conditions for a candidate schedule to be optimal. Reminiscent of the approach used in the case of continuous speeds [Bar+13], we then interpret these conditions in the following geometric manner: Each job j will be associated with a linear function $D_j^{\alpha_j}(t)$, which we call *dual line*. This dual line has a slope of $-d_j$ and passes through the point (r_j, α_j). Here t is time, α_j is the dual variable associated with the primal constraint that all the work from job j must be completed, r_j is the release time of job j, and d_j is the density of job j. Given such an α_j for each job j, one can obtain an associated schedule as follows: At every time t, the job j being processed is the one whose dual line is the highest at that time, and the speed of the processor depends solely on the height of this dual line at that time.

Figure 5.1a shows the dual lines for four different jobs on a processor with three speed levels. The horizontal axis is time. The two horizontal dashed lines labeled by C_2 and C_3 represent the heights where the speed will transition between the lowest speed level and the middle speed level, and the middle speed level and the highest speed level, respectively. Note that these lines only

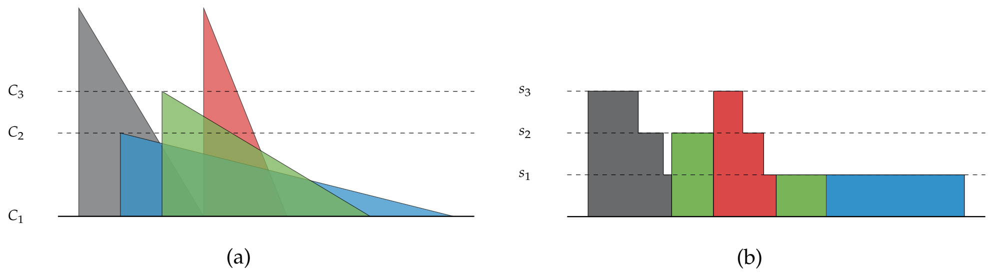


Figure 5.1: Dual lines for a 4-job instance, and the associated schedule.

depend on the speeds and powers of the corresponding level (and not on the jobs). Figure 5.1b shows the associated schedule.

Algorithmic Idea By complementary slackness, a schedule corresponding to a collection of α_j's is optimal if and only if it processes exactly p_j units of work for each job j. Thus we can reduce finding an optimal schedule to finding values for these dual variables with this property. We do so via a primal-dual algorithm that raises the dual α_j variables in an organized way. We iteratively consider the jobs by decreasing density. In iteration i, we construct the optimal schedule S_i for the i most dense jobs from the optimal schedule S_{i-1} for the $i-1$ most dense jobs. We raise the new dual variable α_i from 0 until the associated schedule processes p_i units of work from job i. At some point, raising the dual variable α_i may cause the dual line for i to "affect" the dual line of a previous job j in the sense that α_j must be raised as α_i is raised in order to maintain the invariant that job j is fully processed. Intuitively one might think of "affection" as meaning that the dual lines intersect (this is not strictly correct, but might be a useful initial geometric interpretation to gain intuition). More generally, this affection relation can be transitive in the sense that raising the dual variable α_j may in turn affect another job, and so forth.

Tracking of Affections The algorithm maintains an affection tree rooted at i that describes the affection relationship between jobs, and maintains for each edge in the tree a variable describing the relative rates at which the two incident jobs must be raised in order to maintain the invariant that the proper amount of work is processed for each job. Thus this tree describes the rates that the dual variables of old jobs must be raised as the new dual variable α_i is raised at a unit rate.

In order to discretize the raising of the dual lines, we define four types of events that cause a modification to the affection tree when

- a pair of jobs either begin or cease to affect each other,
- a job either starts or stops using a certain speed level,
- the rightmost point on a dual line crosses another job's release time, or
- enough work is processed on the new job *i*.

During an iteration, the algorithm repeatedly computes when the next such event will occur, raises the dual lines until this event, and then computes the new affection tree. Iteration *i* completes when job *i* has processed enough work.

Correctness & Running Time Its correctness follows from the facts that (a) the affection graph is a tree, (b) this affection tree is correctly computed, (c) the four aforementioned events are exactly the ones that change the affection tree, and (d) the next such event is correctly computed by the algorithm. We bound the running time by bounding the number of events that can occur, the time required to calculate the next event of each type, and the time required to recompute the affection tree after each event.

5.4 Structural Properties via Primal-Dual Formulation

In the following, we give an integer linear programming (ILP) description of our problem. To this end, we start by showing that time can be divided into discrete time slots such that, without loss of generality, during each time slot the processor runs at constant speed and processes at most one job. Note that the resulting time slots may be arbitrarily small, yielding an ILP with many variables and, thus, rendering a direct solution approach less attractive. However, we are actually not interested in solving this ILP directly. Instead, we merely strive to use it and its dual in order to obtain some simple structural properties of an optimal schedule.

$$\begin{aligned} \min\quad & \sum_{j\in\mathcal{J}}\sum_{t=r_j}^{T}\sum_{i=1}^{k} x_{jti}(P_i + s_i d_j(t - r_j + 1/2)) \\ \text{s.t.}\quad & \sum_{t=r_j}^{T}\sum_{i=1}^{k} x_{jti}\cdot s_i \geq p_j \qquad \forall j \\ & \sum_{j\in\mathcal{J}}\sum_{i=1}^{k} x_{jti} \leq 1 \qquad \forall t \\ & x_{jti}\in\{0,1\} \quad \forall j,t,i \end{aligned}$$

(a) ILP formulation of our scheduling problem.

$$\begin{aligned} \max\quad & \sum_{j\in\mathcal{J}} p_j\alpha_j - \sum_{t=1}^{T}\beta_t \\ \text{s.t.}\quad & \beta_t \geq \alpha_j s_i - P_i \\ & \qquad -s_i d_j(t - r_j + 1/2) \\ & \qquad \forall j,t,i\colon t\geq r_j \\ & \alpha_j \geq 0 \quad \forall j \\ & \beta_t \geq 0 \quad \forall t \end{aligned}$$

(b) Dual program of the ILP's relaxation.

Figure 5.2

Discretizing Time Consider $\varepsilon > 0$ and let $T\in\mathbb{N}$ be such that $T\varepsilon$ is an upper bound on the completion time of non-trivial[3] schedules (e.g., $T\varepsilon \geq \max_j(r_j + \sum_j p_j/s_1)$). Given a fixed problem instance, there is only a finite number of jobs and, without loss of generality, an optimal schedule performs only a finite number of speed switches and preemptions. Thus, we can choose $\varepsilon > 0$ such that

(a) any release time r_j is a multiple of ε,

(b) an optimal schedule can use constant speed during $[(t-1)\varepsilon, t\varepsilon)$, and

(c) there is at most one job processed during $[(t-1)\varepsilon, t\varepsilon)$.

We refer to an interval $[(t-1)\varepsilon, t\varepsilon)$ as the t-th time slot. By rescaling the problem instance we can further assume that time slots are of length one (scale r_j by $1/\varepsilon$ and scale s_i as well as P_i by ε).

ILP & Dual Program Let the indicator variable x_{jti} denote whether job j is processed in slot t at speed s_i. Moreover, let T be the aforementioned upper bound on the total number of time slots. This allows us to model our scheduling problem via the ILP given in Figure 5.2a. The first set of constraints ensures that all jobs are completed, while the second set of constraints ensures that the

[3]A non-trivial schedule is one that never runs at speed 0 when there is work remaining.

processor runs at constant speed and processes at most one job in each time slot.

In order to use properties of duality, we consider the relaxation of the above ILP. It can easily be shown that any optimal schedule will always use highest density first as its scheduling policy. This, together with the way we chose ε to construct the time slots, yields that the value of an optimal solution to the LP cannot be less than the value of an optimal solution to the ILP. After considering this relaxation and taking the dual, we get the dual program shown in Figure 5.2b.

The complementary slackness conditions of our primal-dual program are

$$\alpha_j > 0 \quad \Rightarrow \quad \sum_{t=r_j}^{T} \sum_{i=1}^{k} x_{jti} \cdot s_i = p_j, \tag{5.1}$$

$$\beta_t > 0 \quad \Rightarrow \quad \sum_{j \in \mathcal{J}} \sum_{i=1}^{k} x_{jti} = 1, \tag{5.2}$$

$$x_{jti} > 0 \quad \Rightarrow \quad \beta_t = \alpha_j s_i - P_i - s_i d_j (t - r_j + 1/2). \tag{5.3}$$

By complementary slackness, any pair of feasible primal-dual solutions that fulfills these conditions is optimal. We will use this in the following to find a simple way to characterize optimal schedules.

A simple but important observation is that we can write the last complementary slackness condition as $\beta_t = s_i(\alpha_j - d_j(t - r_j + 1/2)) - P_i$. Using complementary slackness, the function $t \mapsto \alpha_j - d_j(t - r_j)$ can help to characterize optimal schedules. The following definitions capture a parametrized version of these job-dependent functions and state how they imply a corresponding (not necessarily feasible) schedule.

Definition 5.1 (Dual Lines and Upper Envelope)**.** For a value $a \geq 0$ and a job j we denote the linear function $D_j^a \colon [r_j, \infty) \to \mathbb{R}, t \mapsto a - d_j(t - r_j)$ as the *dual line* of j with offset a.

Given a job set $H \subseteq \mathcal{J}$ and corresponding dual lines $D_j^{a_j}$, we define the *upper envelope* of H by the upper envelope of its dual lines. That is, the upper envelope of H is a function $\mathrm{UE}_H \colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}, t \mapsto \max_{j \in H}(D_j^{a_j}(t), 0)$. We omit the job set from the index if it is clear from the context.

For technical reasons, we will have to consider the discontinuities in the upper

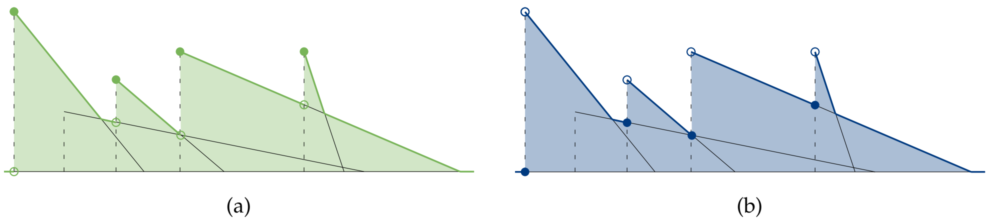


Figure 5.3: Illustration of (a) the upper and (b) the left upper envelope.

envelope separately. Thus, let us also define the left upper envelope below.

Definition 5.2 (Left Upper Envelope and Discontinuity)**.** Given a job set $H \subseteq \mathcal{J}$ and upper envelope of H, UE_H, we define the *left upper envelope* at a point t as the limit of UE_H as we approach t from the left. That is, the left upper envelope of H is a function $\mathrm{LUE}_H \colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}, t \mapsto \lim_{t' \to t^-} \mathrm{UE}_H(t')$. Note that an equivalent definition of the left upper envelope is $\mathrm{LUE}_H(t) = \max_{j \in H : r_j < t}(D_j^{a_j}(t), 0)$.

We say that a point t is a *discontinuity* if UE has a discontinuity at t. Note that this implies that $\mathrm{UE}(t) \neq \mathrm{LUE}(t)$.

An illustration of the upper and left upper envelope is given in Figure 5.3

For the following definition, let us denote $C_i := \frac{P_i - P_{i-1}}{s_i - s_{i-1}}$ for $i \in [k+1]$ as the i-th *speed threshold*. We use it to define the speeds at which jobs are to be scheduled. It will also be useful to define $\hat{C}(x) = \min_{i \in [k+1]} \{\, C_i \mid C_i > x \,\}$ and $\check{C}(x) = \max_{i \in [k+1]} \{\, C_i \mid C_i \leq x \,\}$.

Definition 5.3 (Line Schedule)**.** Consider dual lines $D_j^{a_j}$ for all jobs. The corresponding *line schedule* schedules job j in all intervals $I \subseteq [r_j, \infty)$ of maximal length in which j's dual line is on the upper envelope of all jobs (i.e., $\forall t \in I \colon D_j^{a_j}(t) = \mathrm{UE}(t)$). The speed of a job j scheduled at time t is s_i, with i such that $C_i = \check{C}(D_j^{a_j}(t))$.

See Figure 5.1 for an example of a line schedule. Together with the complementary slackness conditions, we can now easily characterize optimal line schedules.

Lemma 5.4. *Consider dual lines $D_j^{a_j}$ for all jobs. The corresponding line schedule is optimal with respect to fractional weighted flow plus energy if it schedules exactly p_j units of work for each job j.*

Proof. Consider the solution x to the ILP induced by the line schedule. We use the offsets a_j of the dual lines to define the dual variables $\alpha_j := a_j + \frac{1}{2}d_j$. For $t \in \mathbb{N}$, set $\beta_t := 0$ if no job is scheduled in the t-th slot and $\beta_t := s_i D_j^{\alpha_j}(t) - P_i$ if job j is scheduled at speed s_i during slot t. It is easy to check that x, α, and β are feasible and that they satisfy the complementary slackness conditions. Thus, the line schedule must be optimal. □

5.5 Computing an Optimal Schedule

Recall the algorithmic idea roughly sketched in Section 5.3: we aim at a primal dual algorithm that raises the dual variables in an organized way. Lemma 5.4 provides us with a hint of how to organize this raising process. Raising the dual variables corresponds to (geometrically) raising the dual lines. Given Lemma 5.4, one could try to simply raise the dual line of a job i, leaving the remaining dual lines static for the moment. At some point, its dual line will claim enough time on the upper envelope to be fully processed. However, in doing so, we may affect (i.e., reduce) the time windows of other (already scheduled) jobs. Thus, while raising i's dual line, we must keep track of any affected jobs and ensure that they remain fully scheduled. Sections 5.5.1 to 5.5.3 formalize this idea by defining affections and by structuring them in such a way that we can efficiently keep track of them.

Describing the Raising Process Within iteration i of the algorithm, τ will represent how much we have raised α_i. We can think of τ as the time parameter for this iteration of the algorithm (not time in the original problem description but with respect to raising dual-lines). To simplify notation, we do not index variables by the current iteration of the algorithm. In fact, note that every variable in our description of the algorithm may be different at each iteration of the algorithm, e.g., for some job j, $\alpha_j(\tau)$ may be different at the i-th iteration than at the $i+1$-st iteration. To further simplify notation, we use D_j^τ to denote the dual line of job j with offset $\alpha_j(\tau)$. Similarly, we use UE^τ to denote the upper envelope of all dual lines D_j^τ for $j \in [i]$ and S_i^τ to denote the corresponding line schedule. Prime notation generally refers to the rate of change of a variable with respect to τ. To lighten notation even further, we drop τ from variables if its value is clear from the context.

5.5.1 Affected Jobs

We start by formally defining a relation capturing the idea of jobs affecting each other while being raised.

Definition 5.5 (Affection)**.** Consider two different jobs j and j'. We say job j *affects* job j' at time τ if raising (only) the dual line D_j^τ would decrease the processing time of j' in the corresponding line schedule.

We write $j \to j'$ to indicate that j affects j' (and refer to the parameter τ separately, if not clear from the context). Similarly, we write $j \nrightarrow j'$ to state that j does not affect j'. Let us gather some observations about the cases in which jobs affect other jobs. See Figures 5.4 and 5.5 for illustrations.

Observation 5.6. Given jobs j and j' with $j \to j'$, their dual lines must intersect on the upper envelope, or on the left upper envelope at a discontinuity. That is, if t is the intersection point of j and j', we have either $D_j^\tau(t) = D_{j'}^\tau(t) = \mathrm{UE}^\tau(t)$, or $D_j^\tau(t) = D_{j'}^\tau(t) = \mathrm{LUE}^\tau(t)$ and t is a discontinuity. Further, there must be some $\varepsilon > 0$ such that j' is run in either $(t-\varepsilon, t)$ or $(t, t+\varepsilon)$.

Observation 5.7. For $t \in \mathbb{R}_{\geq 0}$ consider the maximal set H_t of jobs that intersect the upper envelope at t and define $H_t^- := H_t \cap \{\, j \mid r_j < t \,\}$. Let $\check{\imath} \in H_t$ denote the job of lowest density and let $\hat{\imath} \in H_t^-$ denote the job of highest density in the corresponding sets (assuming the sets are nonempty). The following hold:

(a) For all $j \in H_t \setminus \{\, \check{\imath} \,\}$ we have $j \to \check{\imath}$.

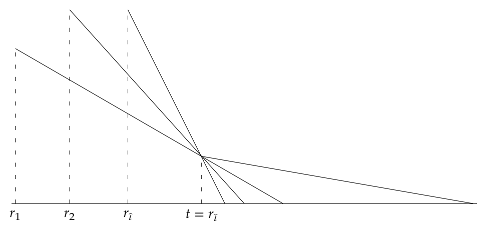
(b) For all $j \in H_t^- \setminus \{\, \hat{\imath} \,\}$ we have $j \to \hat{\imath}$.

(c) For all $j \in H_t$ and $j' \in H_t \setminus \{\, \check{\imath}, \hat{\imath} \,\}$ we have $j \nrightarrow j'$.

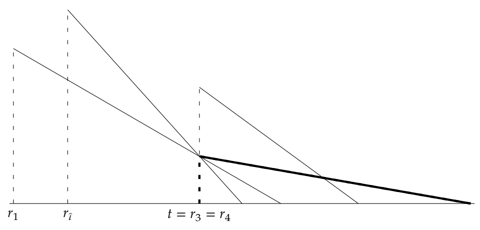
Observation 5.8. For $t \in \mathbb{R}_{\geq 0}$ consider the maximal set H_t of jobs that intersect the left upper envelope at t where t is a discontinuity. Define $H_t^- := H_t \cap \{\, j \mid r_j < t \,\}$. Let $\hat{\imath} \in H_t^-$ denote the job of highest density in H_t^- (assuming it is nonempty). The following hold:

(a) For all $j \in H_t^- \setminus \{\, \hat{\imath} \,\}$ we have $j \to \hat{\imath}$.

(b) For all $j \in H_t$ and $j' \in H_t \setminus \{\, \hat{\imath} \,\}$ we have $j \nrightarrow j'$.



(a) Affections: $\{1,2,\hat{\imath}\} \to \{\check{\imath}\}$, $\{1,2\} \to \{\hat{\imath}\}$, and $\{1,2,\hat{\imath},\check{\imath}\} \twoheadrightarrow \{1,2\}$.



(b) Affections: $1 \to \hat{\imath}$ and $\{1,\hat{\imath},3\} \twoheadrightarrow \{1,3\}$. Job 3 is denoted by the thick line.

Figure 5.4: Illustration of (a) Observation 5.7 and (b) Observation 5.8.

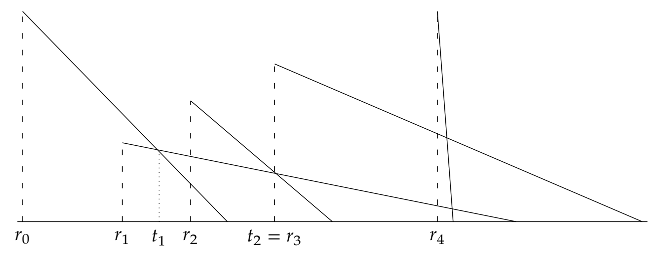


Figure 5.5: Observation 5.10. Both the upper envelope and the left upper envelope cases: note that $D_0^\tau(t') < \mathrm{LUE}^\tau(t') \leq \mathrm{UE}^\tau(t')$ for all $t' > t_1$, and $D_2^\tau(t') < \mathrm{LUE}^\tau(t') \leq \mathrm{UE}^\tau(t')$ for all $t' > t_2$.

Observation 5.9. No job $j \in [i-1]$ can intersect a job j' of lower density at its own release time r_j.

Observation 5.10. Given jobs j and j' with $d_j > d_{j'}$ and that intersect on the upper envelope or left upper envelope at point t, we have that $\mathrm{UE}^\tau(t') \geq \mathrm{LUE}^\tau(t') \geq D_{j'}^\tau(t') > D_j^\tau(t')$, for all $t' > t$.

Structuring the Affection Equipped with the aforementioned observations, we continue to show several structural properties about how different jobs can affect each other. Remember that we assume jobs to be ordered by decreasing density (see Section 5.2) and fix a job i. In the following, we study how the raising of job i can affect the jobs in $\{1, 2, \ldots, i-1\}$. We will use our insights in Section 5.5.2 to prove that the graph induced by the affection relation forms a tree.

Define the level sets $\mathcal{L}_0 := \{i\}$ and $\mathcal{L}_l := \{j \mid \exists j_- \in \mathcal{L}_{l-1} : j_- \to j\} \setminus \bigcup_{l'=0}^{l-1} \mathcal{L}_{l'}$ for an integer $l \geq 1$. Intuitively, a job j is in level set $\mathcal{L}_l$ if and only if the "shortest path" by which it is affected by i has length l. With this notation, we are now ready to prove the following lemmas.

Lemma 5.11. *Consider two jobs $j_0 \in \mathcal{L}_l$ and $j_+ \in \mathcal{L}_{l+1}$ with $j_0 \to j_+$. Then job j_+ has a larger density than job j_0. That is, $d_{j_+} > d_{j_0}$.*

Proof. We prove the lemma's statement by induction. The base case $l = 0$ is trivial, as i has the lowest density of all jobs and, by construction, $\mathcal{L}_0 = \{ i \}$. Now consider the case $l \geq 1$ and let $j_- \in \mathcal{L}_{l-1}$ be such that $j_- \to j_0$. By the induction hypothesis, we have $d_{j_0} > d_{j_-}$. For the sake of a contradiction, assume $d_{j_+} < d_{j_0}$. Let t_1 denote the intersection point of j_0 and j_-, and let t_2 denote the intersection point of j_0 and j_+. By Observation 5.6, these intersection points lie on the upper envelope or the left upper envelope. We consider three cases:

Case $t_1 > t_2$: Because of $t_1 > t_2$, the assumption $d_{j_0} > d_{j_+}$ implies $D^{\tau}_{j_0}(t_1) < D^{\tau}_{j_+}(t_1) \leq \mathrm{LUE}^{\tau}(t_1) \leq \mathrm{UE}^{\tau}(t_1)$ by Observation 5.10. This contradicts Observation 5.6.

Case $t_1 < t_2$: Because of $t_2 > t_1$, the induction hypothesis $d_{j_0} > d_{j_-}$ implies, by Observation 5.10, $D^{\tau}_{j_0}(t_2) < D^{\tau}_{j_-}(t_2) \leq \mathrm{LUE}^{\tau}(t_2) \leq \mathrm{UE}^{\tau}(t_2)$. This contradicts Observation 5.6.

Case $t_1 = t_2$: First note that this intersection point must lie on the upper envelope since otherwise it would lie on the left upper envelope at a discontinuity and, by Observation 5.8, $j_0 \nrightarrow j_+$. Further, because $j_0 \neq i$ and $d_{j_0} > d_{j_+}$, Observation 5.9 implies $r_{j_0} < t_1$. Together with $d_{j_+} < d_{j_0}$ and $j_0 \to j_+$, this implies that j_+ has minimal density among all jobs intersecting the upper envelope in t_1 (by Observation 5.7). We get $j_- \to j_+$ and, thus, $j_+ \in \mathcal{L}_l$. This contradicts j_+ being a level $l + 1$ node. □

Lemma 5.12. *Given two level l jobs $j_1, j_2 \in \mathcal{L}_l$, we have $j_1 \nrightarrow j_2$ and $j_2 \nrightarrow j_1$.*

Proof. The statement is trivial for $l = 0$, as $\mathcal{L}_0 = \{ i \}$. For $l \geq 1$ consider $j_1, j_2 \in \mathcal{L}_l$ and assume, for the sake of a contradiction, that $j_1 \to j_2$ (the case $j_2 \to j_1$ is symmetrical). Let $\iota_1, \iota_2 \in \mathcal{L}_{l-1}$ with $\iota_1 \to j_1$ and $\iota_2 \to j_2$. By Lemma 5.11 we have $d_{\iota_1} < d_{j_1}$ and $d_{\iota_2} < d_{j_2}$. Let t_0 denote the intersection point of j_1 and j_2, t_1 the intersection point of j_1 and ι_1, and t_2 the intersection point of j_2 and ι_2. Analogously to the proof of Lemma 5.11, one can see that $t_0 = \min(t_1, t_2)$ (as otherwise at least one of these intersection points would not lie on the (left) upper envelope). We distinguish the following cases:

Case $t_0 = t_1 < t_2$: First note that t_1 and t_0 cannot lie on the left upper envelope at a discontinuity, since by Observation 5.8 either $j_1 \nrightarrow j_2$ or $\iota_1 \nrightarrow j_1$. So, by Observation 5.6, t_0 and t_1 lie on the upper envelope. Job j_2 must have minimal density among all jobs intersecting the upper envelope at t_1, as otherwise its intersection point with ι_2 cannot lie on the upper envelope. But then, by Observation 5.7, we have $\iota_1 \rightarrow j_2$. Together with Lemma 5.11 this implies $d_{j_2} > d_{\iota_1}$, contradicting the minimality of j_2's density.

Case $t_0 = t_2 < t_1$: By Observation 5.6 j_1, j_2 and ι_2 either lie on the left upper envelope or the upper envelope. Assume they are on the left upper envelope. Note that since ι_2 and j_1 intersect at t_0, and $t_1 > t_0$, it must be that $\iota_1 \neq \iota_2$ and therefore $l \geq 2$ in this case. Also, since $t_1 > t_0$ is a point where j_1 is on the upper envelope, it must be that j_1 is less dense than ι_2. However, this implies that ι_2 is not on the left upper envelope or upper envelope to the right of t_0. Since it is not the root ($l \geq 2$) there must be some point $t < t_0$ such that ι_2 intersects a less dense job on the (left) upper envelope (its parent). This contradicts ι_2 being on the left upper envelope at t_0. If instead j_1, j_2 and ι_2 lie on the upper envelope the same argument used in the first case applies.

Case $t_0 = t_1 = t_2$: The same argument as in the first case shows that these points do not lie on the left upper envelope at a discontinuity but must lie on the upper envelope. Without loss of generality, assume $d_{j_1} > d_{j_2}$. We get $r_{j_1} < t_1$ (Observation 5.9). With $d_{j_2} > d_{\iota_2}$ and Observation 5.7 this implies $\iota_2 \nrightarrow j_2$, contradicting the definition of ι_2. ☐

Lemma 5.13. *A level l job cannot be affected by more than one job of a lower level.*

Proof. The statement is trivial for $l \in \{0, 1\}$. Thus, consider a job $j \in \mathcal{L}_l$ for $l \geq 2$ and let j_1 and j_2 be two different lower level jobs with $j_1 \rightarrow j$ and $j_2 \rightarrow j$. By definition, both j_1 and j_2 must be level $l - 1$ jobs. Also, similar to previous proofs, we can see that all three jobs must intersect at the same point t on the upper envelope or left upper envelope. Let us first assume they intersect at the upper envelope. Observation 5.7 implies that j has maximal density of all jobs intersecting the upper envelope at t (as otherwise j can be affected by neither j_1 nor j_2, both having a lower density). Consider the lowest density job $\breve{\iota}$ intersecting the upper envelope at t. By Observation 5.7, at least one of j_1 or

j_2 must affect $\check{\imath}$. Assume, without loss of generality, it is j_1. This implies that $\check{\imath}$ has level $l' \leq l$. Actually, we must have $l' < l$, because otherwise $d_{\check{\imath}} < d_{j_1}$ would contradict Lemma 5.11. Similarly, $l' = l - 1$ would contradict Lemma 5.12. Thus, we have $l' \leq l - 2$. But since we have $\check{\imath} \rightarrow j$, we get a contradiction to j being a level l node.

Now assume that all three jobs intersect at a discontinuity of the left upper envelope. Observation 5.8 tells us that j must be the most dense job intersecting at t. Assume without loss of generality that $d_{j_1} < d_{j_2}$. Then, by Observation 5.10, j_2 is not on the (left) upper envelope to the right of t. However, since it is not the root ($l \geq 2$), there must be some less dense job $\check{\imath}$ that intersects j_2 on the (left) upper envelope to the left of t (its parent). This contradicts j_2 being on the left upper envelope at t. □

Lemma 5.14. *Consider two nodes $j_1 \in \mathcal{L}_{l_1}$ and $j_2 \in \mathcal{L}_{l_2}$ with $l_2 - l_1 \geq 2$. Then we must have $j_2 \nrightarrow j_1$.*

Proof. For the sake of a contradiction, assume $j_2 \rightarrow j_1$ and let j denote a level $l_2 - 1$ node with $j \rightarrow j_2$. Similar to the previous proofs, we can see that all three jobs j_1, j_2, and j must intersect the upper envelope or left upper envelope at a common intersection point t. In the first case, assume this is on the upper envelope. Because of $d_{j_2} > d_j$ and $j \rightarrow j_2$, we get that $r_{j_2} < t$ and that j_2 has maximal density among all jobs intersecting the upper envelope at t and having a release time before t (Observation 5.9 and Observation 5.7). We get $j_1 \rightarrow j_2$, contradicting j_2 being of at least level $l_1 + 2$.

In the case when they intersect at a discontinuity of the left upper envelope, Observation 5.8 implies either $j_2 \nrightarrow j_1$ or $j \nrightarrow j_2$, a contradiction. □

Lemma 5.15. *Consider two nodes $j_1 \in \mathcal{L}_{l_1}$ and $j_2 \in \mathcal{L}_{l_2}$ with $l_2 = l_1 + 1$, and $j_1 \rightarrow j_2$. Then, if there exists a node $j_3 \in \mathcal{L}_{l_1}$ such that $j_2 \rightarrow j_3$, it must be that $j_3 = j_1$.*

Proof. For the sake of contradiction, assume there exists a node $j_3 \neq j_1$ such that $j_3 \in \mathcal{L}_{l_1}$ and $j_2 \rightarrow j_3$. First, note that by Lemma 5.11 we have $d_{j_1} < d_{j_2}$. Let t_1 be the intersection of j_1 and j_2, and t_2 be the intersection of j_2 and j_3. There are two cases to consider. In the first case, assume $t_1 = t_2$ and note that the intersection must lie on the upper envelope: if it were on the left upper envelope at a discontinuity, $j_1 \rightarrow j_2$ and $j_2 \rightarrow j_3$ would contradict Observation 5.8. Since

$d_{j_1} < d_{j_2}$, either j_1 or j_3 is the job with lowest density. Then, since $t_1 = t_2$, by Observation 5.7 either $j_1 \to j_3$ or $j_3 \to j_1$. Both cases contradict Lemma 5.12 since $j_1, j_3 \in \mathcal{L}_{l_1}$.

In the second case, assume $t_1 \neq t_2$. If $t_1 < t_2$, then, since $d_{j_1} < d_{j_2}$ by Observation 5.10, $D^{\tau}_{j_2}(t') < \mathrm{LUE}^{\tau}(t') \leq \mathrm{UE}^{\tau}(t')$ for all $t' > t_1$ which contradicts j_2 being on the (left) upper envelope at t_2. For the case $t_1 > t_2$, a similar argument shows that j_2 must be the least dense job that intersects the upper envelope at t_2, as otherwise $D^{\tau}_{j_2}(t_1) < \mathrm{LUE}^{\tau}(t_1) \leq \mathrm{UE}^{\tau}(t_1)$. If the jobs meet on the upper envelope at t_2, Observation 5.7 yields $j_3 \to j_2$. Together with $j_1 \to j_2$ and $j_3 \neq j_1$, this contradicts Lemma 5.13. If the jobs meet on the left upper envelope at t_2, we see similarly to previous proofs that j_3 is not the root, cannot be processed to the right of t_2 (since j_2 is less dense), and its less dense parent muss intersect it to the left of t_2. But then j_2 cannot be on the left upper envelope at t_2, a contradiction. □

5.5.2 Affection Tree

Lemmas 5.11 to 5.15 give us strong structural information about how raising the dual line of job i affects jobs in $\{1, 2, \ldots, i-1\}$. In particular, the affection relation naturally defines a graph on the jobs as follows:

Definition 5.16 (Affection Tree)**.** Let $G_i(\tau)$ be the directed graph induced by the affection relation on jobs $1, \ldots, i$. Then the *affection tree* is an undirected graph $A_i(\tau) = (V_i(\tau), E_i(\tau))$ where $j \in V_i(\tau)$ if and only if j is reachable from i in $G_i(\tau)$, and for $j_1, j_2 \in V_i(\tau)$ we have $(j_1, j_2) \in E_i(\tau)$ if and only if $j_1 \to j_2$ or $j_2 \to j_1$.

In the following lemma, we prove that A_i is indeed a tree rooted at i such that all children of a node j are of higher density than j itself.

Lemma 5.17. *Let A_i be the (affection) graph of Definition 5.16. Then A_i is a tree, and if we root A_i at i, then for any parent and child pair $(\iota_j, j) \in G$ it holds that $d_{\iota_j} < d_j$.*

Proof. Assume, for the sake of contradiction, that there exists a cycle C in G. Let v be a node in C that belongs to the highest level set, say $\mathcal{L}_{l_1}$. Note that such a v is unique since otherwise there would be two nodes in the same level with at least one having an affection to the other contradicting Lemma 5.12. Let v_1, v_2 be the neighbors of v in C and $v_3 \in \mathcal{L}_{l_1-1}$ be the node such that $v_3 \to v$ (Note that it may be $v_3 = v_1$ or $v_3 = v_2$). Note that by Lemma 5.14 we also have $v_1, v_2 \in \mathcal{L}_{l_1-1}$. By Lemma 5.13, either $v_1 \nrightarrow v$ or $v_2 \nrightarrow v$. Assume, without loss of generality, this is v_1. Since v_1 is a neighbor of v in C and $v_1 \nrightarrow v$, we have $v \to v_1$. However, this contradicts Lemma 5.15. □

In the remainder, we will always assume $A_i(\tau)$ to be rooted at i and use the notation $(j, j') \in A_i(\tau)$ to indicate that j' is a child of j. The proven tree structure of the affection graph will allow us to easily compute how fast to raise the different dual lines of jobs in A_i (as long as the connected component does not change).

5.5.3 Algorithm Description

We are finally (almost) ready to state our algorithm. Our algorithm will use the affection tree to track the jobs affected by the raising of the current job i and compute corresponding raising speeds for each of these jobs. This raising stops, at latest, when job i is completely scheduled. However, there are other events we must consider, like a change in the topology of the affection tree when new nodes are affected. The intuition for each event is comparatively simple (see Definition 5.18), but their formalization is quite technical, requiring us to explicitly label the start and ending points of each single execution interval of each job. To do so, we introduce the following *interval notation*.

Interval Notation Let $\hat{r}_1, \dots, \hat{r}_n$ denote the n release times in non-decreasing order. We define Ψ_j as a set of indices with $q \in \Psi_j$ if and only if job j is run between $\hat{r}_q$ and $\hat{r}_{q+1}$ (or after $\hat{r}_n$ for $q = n$). Further, let $x_{\ell,q,j}$ denote the time that the interval corresponding to q begins and $x_{r,q,j}$ denote the time that the interval ends. Let $s_{\ell,q,j}$ denote the speed at which j is running at the left endpoint corresponding to q and $s_{r,q,j}$ denote the speed j is running at the right endpoint. Let $q_{\ell,j}$ be the smallest and $q_{r,j}$ be the largest indices of Ψ_j, i.e., the indices of the first and last execution intervals of j.

Let the indicator variable $y_{r,j}(q)$ denote whether $x_{r,q,j}$ occurs at a release point. Similarly, $y_{\ell,j}(q) = 1$ if $x_{\ell,q,j}$ occurs at r_j, and 0 otherwise. Lastly, $\chi_j(q)$ is 1 if q is not the last interval in which j is run, and 0 otherwise.

We define $\rho_j(q)$ to be the last interval of the uninterrupted block of intervals starting at q, i.e., for all $q' \in \{q+1, \dots, \rho_j(q)\}$, we have that $q' \in \Psi_j$ and $x_{r,q'-1,j} = x_{\ell,q',j}$, and either $\rho_j(q) + 1 \notin \Psi_j$ or $x_{r,\rho_j(q),j} \neq x_{\ell,\rho_j(q)+1,j}$.

Note that, as the line schedule changes with τ, so does the set of intervals corresponding to it, therefore we consider variables relating to intervals to be functions of τ as well (e.g., $\Psi_j(\tau)$, $x_{\ell,q,j}(\tau)$, etc.).

Events & Algorithm Given this notation, we now define four different types of events that force us to recalculate the rate at which the dual lines are raised. We assume that between τ and the next event τ_0, we raise each dual line at a constant rate. More formally, we fix τ and for $j \in [i]$ and $u \geq \tau$ let $\alpha_j(u) = \alpha_j(\tau) + (u - \tau)\alpha_j'(\tau)$.

Definition 5.18 (Event). For $\tau_0 > \tau$, we say that an *event occurs at* τ_0 if there exists $\varepsilon > 0$ such that at least one of the following holds for all $u \in (\tau, \tau_0)$ and $v \in (\tau_0, \tau_0 + \varepsilon)$:

- The affection tree changes, i.e., $A_i(u) \neq A_i(v)$. This is called an *affection change event*.
- The speed at the border of some interval of some job changes. That is, there exists a $j \in [i]$ and a $q \in \Psi_j(\tau)$ such that either $s_{\ell,q,j}(u) \neq s_{\ell,q,j}(v)$ or $s_{r,q,j}(u) \neq s_{r,q,j}(v)$. This is called a *speed change event*.
- The last interval in which job i is run changes from ending before the release time of some other job to ending at the release time of that job. That is, there exists $j \in [i-1]$ and $q \in \Psi_i(\tau)$ such that $x_{r,q,i}(u) < r_j$ and $x_{r,q,i}(v) = r_j$. This is called a *simple rate change event*.
- Job i completes enough work, i.e., $p_i(u) < p_i < p_i(v)$. This is called a *job completion event*.

A formal description of the algorithm can be found in Listing 5.1.

5.6 Correctness of the Algorithm

In the following, we focus on proving the correctness of the previously described algorithm. Throughout this subsection, we assume the iteration and

```
for each job $i$ from 1 to $n$:
    while $p_i(\tau) < p_i$:        {job i not yet fully processed in current schedule }
        for each job $j \in A_i(\tau)$:
            calculate $\delta_{j,i}(\tau)$        {see Equation (5.6)}
        let $\Delta\tau$ be the smallest $\Delta\tau$ returned by any of the subroutines below:
            (a) JobCompletion($S(\tau), i, [\alpha'_1, \alpha'_2, \dots, \alpha'_i]$)            {next job completion}
            (b) AffectionChange($S(\tau), A_i(\tau), [\alpha'_1, \alpha'_2, \dots, \alpha'_i]$)  {next affection change}
            (c) SpeedChange($S(\tau), [\alpha'_1, \alpha'_2, \dots, \alpha'_i]$)                {next speed change}
            (d) RateChange($S(\tau), i, [\alpha'_1, \alpha'_2, \dots, \alpha'_i]$)              {next rate change}
        for each job $j \in A_i(\tau)$:
            raise $\alpha_j$ by $\Delta\tau \cdot \delta_{j,i}$
        set $\tau = \tau + \Delta\tau$
        update $A_i(\tau)$ if needed    {only if Case (b) returns the smallest $\Delta\tau$}
```

Listing 5.1: The algorithm for computing an optimal schedule. See Section 5.6.1 for details on the subroutines.

value of τ to be fixed. Recall that we have to raise the dual lines such that the total work done for any job $j \in [i-1]$ is preserved. To calculate the work processed for j in an interval, we must take into account the different speeds at which j is run in that interval. Note that the intersection of j's dual line with the i-th speed threshold C_i occurs at $t = \frac{\alpha_j - C_i}{d_j} + r_j$. Therefore, the work done by a job $j \in [i]$ is given by

$$\begin{aligned} p_j = & \sum_{q \in \Psi_j} s_{\ell,q,j} \left(\frac{\alpha_j - \check{C}(D_j^\tau(x_{\ell,q,j}))}{d_j} + r_j - x_{\ell,q,j} \right) \\ & + \sum_{k: s_{\ell,q,j} > s_k > s_{r,q,j}} s_k \left(\frac{\alpha_j - C_k}{d_j} + r_j - \left(\frac{\alpha_j - C_{k+1}}{d_j} + r_j \right) \right) \\ & + s_{r,q,j} \left(x_{r,q,j} - \left(\frac{\alpha_j - \hat{C}(D_j^\tau(x_{r,q,j}))}{d_j} + r_j \right) \right). \end{aligned} \tag{5.4}$$

It follows that the change in the work of job j with respect to τ is

$$p'_j = \sum_{q \in \Psi_j} \left[s_{\ell,q,j} \left(\frac{\alpha'_j}{d_j} - x'_{\ell,q,j} \right) + s_{r,q,j} \left(x'_{r,q,j} - \frac{\alpha'_j}{d_j} \right) \right]. \tag{5.5}$$

For some child j' of j in A_i, let $q_{j,j'}$ be the index of the interval of Ψ_j that begins with the completion of j'. Recall that D_i^τ is raised at a rate of 1 with respect to τ, and for a parent and child (ι_j, j) in the affection tree, the rate of change for α_j with respect to α_{ι_j} used by the algorithm is:

$$\begin{aligned}\delta_{j,\iota_j} :=& \Bigg(1 + y_{\ell,j}(q_{\ell,j})\frac{d_j - d_{\iota_j}}{d_j}\frac{s_{\ell,q_{\ell,j},j} - s_{r,\rho_j(q_{\ell,j}),j}}{s_{r,q_{r,j},j}} \\ &+ \sum_{(j,j')\in A_i}\Bigg((1-\delta_{j',j})\frac{d_j - d_{\iota_j}}{d_{j'} - d_j}\frac{s_{\ell,q_{j,j'},j}}{s_{r,q_{r,j},j}} + \frac{d_j - d_{\iota_j}}{d_j}\frac{s_{\ell,q_{j,j'},j} - s_{r,\rho(q_{j,j'}),j}}{s_{r,q_{r,j},j}}\Bigg)\Bigg)^{-1}. \end{aligned} \tag{5.6}$$

We prove in Lemma 5.21 that these rates are work-preserving for all jobs $j \in [i-1]$. Note that the algorithm actually uses $\delta_{j,i}$ which we can compute by taking the product of the $\delta_{k,k'}$ over all edges (k,k') on the path from j to i. Similarly, we can compute $\delta_{j,j'}$ for all $j, j' \in A_i$.

Observation 5.19. Since, by Lemma 5.17, parents in the affection tree are always of lower-density than their children, and since dual lines are monotonically decreasing, we have that $\delta_{\iota_j,j} \le 1$. Therefore, intersection points on the upper envelope can never move towards the right as τ is increased.

The following lemma states how fast the borders of the various intervals change with respect to the change in τ.

Lemma 5.20. *Consider any job $j \in A_i$ whose dual line gets raised at a rate of $\delta_{j,i}$.*

(a) For an interval $q \in \Psi_j$, if $y_{\ell,j}(q) = 1$, then $x'_{\ell,q,j} = 0$.

(b) For an interval $q \in \Psi_j$, if $\chi_j(q) = 1$, then $x'_{r,q,j} = 0$.

(c) Let (j,j') be an edge in the affection tree and let q_j and $q_{j'}$ denote the corresponding intervals for j and j'. Then $x'_{\ell,q_j,j} = x'_{r,q_{j'},j'} = -\frac{\alpha'_j - \alpha'_{j'}}{d_{j'} - d_j}$. Note that this captures the case $q \in \Psi_{j'}$ with $\chi_{j'}(q) = 0$ and $j' \neq i$.

(d) For an interval $q \in \Psi_i$, if $\chi_i(q) = 0$, then $x'_{r,q,i} = 0$ or $x'_{r,q,i} = 1/d_i$.

Proof.

(a) Note that since $y_{\ell,j}(q) = 1$, this implies that $x_{\ell,q,r} = r_j$. Since by Observation 5.19 intersection points can only move towards the left but by definition D_j^τ is defined in $[r_j, \infty)$ the statement follows.

(b) Set $t = x'_{r,q,j}$ and let us consider two subcases. In the first case, assume that there exists an $\varepsilon > 0$ such that j is run in $(t, t+\varepsilon)$. Then we must have that $t = x_{r,q,j} = r_{j'}$ for some $j' \neq j$, as otherwise q would not be maximal. This implies $x'_{r,q,j} = 0$.

In the second subcase, assume that there does not exist any $\varepsilon > 0$ such that j is run in $(t, t+\varepsilon)$. This implies that there is some change in the upper envelope at t, which can happen only in the following three cases:

(i) The dual line crosses 0 at t. That is, $\alpha_j - d_j(t - r_j) = 0$.

(ii) The dual line crosses a dual line of smaller slope at t.

(iii) A release time causes a discontinuity on the upper envelope at t.

Note that (i) and (ii) can only happen at the last execution interval of a job, but since $\chi_j(q) = 1$, q is not the last interval in which j is run. In (iii), since $x_{r,q,j} = r_{j'}$ at a discontinuity, $x'_{r,q,j} = 0$ and the statement holds.

(c) Note that since (j, j') is an edge in the affection tree, by Observation 5.6 we have that $D^{\tau}_{j'}$ and D^{τ}_{j} must intersect on the (left) upper envelope. We have $D^{\tau}_{j'}(t) = \alpha_{j'} - d_{j'}(t - r_{j'})$ and $D^{\tau}_{j}(t) = \alpha_j - d_j(t - r_j)$. Therefore, the dual lines for j and j' intersect at

$$t = \frac{\alpha_{j'} + d_{j'} \cdot r_{j'} - \alpha_j - d_j \cdot r_j}{d_{j'} - d_j}, \text{ and its derivative is } \frac{\alpha'_j - \alpha'_{j'}}{d_{j'} - d_j}.$$

Since j is a parent of j', $x_{\ell,q_j,j} = x_{r,q_{j'},j'} = t$ and the result follows.

(d) Note that since job i has the lowest density of all jobs currently considered, its rightmost interval can only stop at a release time of a denser job, or at a point t such that $D^{\tau}_i = 0$. In the first case $x'_{r,q,i} = 0$. In the second case note that $D^{\tau}_i(t) = \alpha_i - d_i(t - r_i)$ intersects 0 at $t = \alpha_i/d_i + r_i$. Taking the derivative with respect to τ yields $x'_{r,q,i} = \alpha'_i/d_i = 1/d_i$, as desired. □

Equation (5.5) defines a system of differential equations. In the following, we first show how to compute a work-preserving solution for this system (in which $p'_j = 0$ for all $j \in [i-1]$) if $\alpha'_i = 1$, and then show that there is only a polynomial number of events and that the corresponding τ values can be easily computed.

Lemma 5.21. *For a parent and child $(\iota_j, j) \in A_i$, set $\alpha'_j = \delta_{j,\iota_j}\alpha'_{\iota_j}$, and for $j' \notin A_i$ set $\alpha_{j'} = 0$. Then $p'_j = 0$ for $j \in [i-1]$.*

Proof. Clearly, by the definition of affection and construction of the affection tree, if $j' \notin A_i$, then by setting $\alpha'_{j'} = 0$ we have that $p'_{j'} = 0$.

For a parent and child $(\iota_j, j) \in A_i$, we proceed by setting $p'_j = 0$ in Equation (5.5) and solving for $\alpha'_j/\alpha'_{\iota_j} = \delta_{j,\iota_j}$. Let $I_{q,j} = \{ q, \dots, \rho_j(q) \}$ if $q \in \Psi_j$ and $\emptyset$ otherwise. We call $I_{q,j}$ a *maximal execution interval* of j if $I_{q-1,j} \cap I_{q,j} = \emptyset$. Let $M = \{ q \in \Psi_j \mid I_{q,j}$ is a maximal execution interval of $j \}$. Clearly $\bigcup_{q \in M} I_{q,j} = \Psi_j$. Let $p'_{j,S}$ where $S \subseteq \Psi_j$ be the rate of change of p_j due to the rate of change of the endpoints of the intervals in S. If j is run at its release time, then $y_{\ell,j}(q_{\ell,j}) = 1$ and, by Observation 5.7, j cannot intersect any of its children at its release time, so by Lemma 5.20

$$p'_{j,I_{q_{\ell,j},j}} = (s_{\ell,q_{\ell,j},j} - s_{r,\rho(q_{\ell,j}),j})\left(\frac{\alpha'_j}{d_j}\right) + s_{r,\rho_j(q_{\ell,j}),j}\left(x'_{r,\rho_j(q_{\ell,j}),j}\right). \tag{5.7}$$

For any other $q \in M$ (including $q_{\ell,j}$ if $y_{\ell,j}(q_{\ell,j}) = 0$), q must begin at the intersection point of j and one of its children. That is, there exists a unique $(j, j') \in A_i$ such that $q = q_{j,j'}$. Therefore, by Lemma 5.20

$$p'_{j,I_{q_{j,j'},j}} = (s_{\ell,q_{j,j'},j} - s_{r,\rho(q_{j,j'}),j})\left(\frac{\alpha'_j}{d_j}\right) + s_{\ell,q_{j,j'},j}\left(\frac{\alpha'_j - \alpha'_{j'}}{d_{j'} - d_j}\right) + s_{r,\rho_j(q_{j,j'}),j}\left(x'_{r,\rho_j(q_{j,j'}),j}\right).$$

For any $q \in M$, we have (Lemma 5.20) that $x'_{r,\rho(q),j} = 0$ if $\rho(q) \neq q_{r,j}$ and $x'_{r,q_{r,j},j} = -\frac{\alpha'_{\iota_j} - \alpha'_j}{d_j - d_{\iota_j}}$. The lemma follows by observing that $p'_j = \sum_{q \in M} p'_{j,I_{q,j}}$, the fact that j must intersect each of its children exactly once on the (left) upper envelope, and that for $(j, j') \in A_i$, we have that $\alpha_{j'}/\alpha_j = \delta_{j',j}$. $\square$

Although it is simple to identify the next occurrence of job completion, speed change, or simple rate change events, it is more involved to identify the next affection change event. Therefore, we provide the following lemma to account for this case.

Lemma 5.22. *An affection change event occurs at time τ_0 if and only if at least one of the following occurs.*

(a) *An intersection point t between a parent and child $(j, j') \in A_i$ becomes equal to r_j. That is, at $\tau_0 > \tau$ such that $D_j^{\tau_0}(r_j) = D_{j'}^{\tau_0}(r_j) = UE^{\tau_0}(r_j)$.*

(b) *Two intersection points t_1 and t_2 on the upper envelope become equal. That is, for $(j_1, j_2) \in A_i$ and $(j_2, j_3) \in A_i$, at $\tau_0 > \tau$ such that there is a t with $D_{j_1}^{\tau_0}(t) = D_{j_2}^{\tau_0}(t) = D_{j_3}^{\tau_0}(t) = UE^{\tau_0}(t)$.*

(c) *An intersection point between j and j' meets the (left) upper envelope at the right endpoint of an interval in which j' was being run. Furthermore, there exists $\varepsilon > 0$ so that for all $\tau \in (\tau_0 - \varepsilon, \tau_0)$, j' was not in the affection tree.*

Proof. It is straightforward to see that whenever (a), (b), or (c) occurs, an affection change event has to take place. Therefore, we focus the rest of the proof on showing the other direction, i.e., any affection change event is a consequence of one of the aforementioned cases.

By definition, any change in the affection tree is built from a sequence of edge additions and edge removals. We therefore will separately consider the cases where an edge is removed or added.

Case 1: An edge between j and j' is removed.

Let τ_0 be a time when an edge between j and j' is removed, and assume that j and j' had an intersection point t. Assume furthermore, without loss of generality, that j is a parent of j'. Therefore, the affection $j \to j'$ ceases to exist at τ_0 (perhaps also $j' \to j$ if it existed). First note that at t, $D_j^{\tau_0}(t) = D_{j'}^{\tau_0}(t)$ must be on the upper envelope or left upper envelope at a discontinuity, otherwise there exists some $\varepsilon > 0$ such that at time $\tau_0 - \varepsilon$ their intersection was not on the upper envelope or the left upper envelope but the edge $j \to j'$ existed, contradicting Observation 5.6. We handle these two cases separately.

Subcase: $D_j^{\tau_0}(t) = D_{j'}^{\tau_0}(t) = \mathrm{UE}^{\tau_0}(t)$.

By Lemma 5.17, we know that it must be the case that $d_{j'} > d_j$. Furthermore, by Observation 5.7, since now it is the first time that $j \not\to j'$, either $r_j = t$ (which is covered in statement (a) of the lemma), or at least three jobs intersect at t. If this is the case, let j'' be the highest density job among the jobs that intersect at t. Note that j'' cannot be j (since $d_{j'} > d_j$) and it can also not be j' (since $j \not\to j'$).

By Observation 5.7, just before τ_0, j' does work to the left of the intersection point (between j and j') and j to the right. But at τ_0, j' cannot do any work directly to the left of t, because of j''. It follows that the interval of j' has disappeared, since to the left of t, j'' is run and to the right of t, j is run. This case is covered in the statement (b) of the lemma.

Subcase: $D_j^{\tau_0}(t) = D_{j'}^{\tau_0}(t) = \mathrm{LUE}^{\tau_0}(t)$ at a discontinuity t.

Note that since the intersection points do not move towards the right as τ increases (by Observation 5.19), the intersection of j and j' was either at t or it was moving to the left towards t for all times during which $j \to j'$. However, since there is a discontinuity at t, there is some job j'' on the upper envelope that is not on the left upper envelope. If the intersection was at t then $j \to j'$ would not be possible. Therefore, there must exist some $\varepsilon > 0$ such that at $\tau_0 - \varepsilon$ the intersection of j and j' was below the curve of j''. This contradicts Observation 5.6. It follows that this subcase cannot occur.

Case 2: A new edge is added to the affection tree.

Let τ_0 be the time when a new edge is added to the affection tree. First note that, without loss of generality, at least one new edge is between two nodes j and j' with (a) j is in the affection tree immediately before τ_0, (b) j' is not in the affection tree immediately before τ_0, and (c) j becomes the parent of j' at τ_0. Indeed, obviously at least one old node j of the affection tree must be involved as a parent in one of the new edges. If all new children j' of such old nodes were in the affection tree immediately before τ_0, there would also be some edge removal at τ_0 (as an additional edge would break the tree property, contradicting Lemma 5.17). This would reduce this case to the previous one.

From the above we get $d_j < d_{j'}$ (j being a parent of j'). Moreover, by Observation 5.6, at τ_0 the intersection point t of j and j' is on the (left) upper envelope and j' is run either to the left or right of t. Since j has a lower density, it must be run to the left of t. This is the case covered by statement (c) of the lemma. □

5.6.1 The Subroutines

The algorithm shown in Listing 5.1 uses four subroutines to compute the time of the next job completion, affection change, speed change, and simple rate change events. Computing these times is relatively easy. The following provides a short sketch for the corresponding computation of each event type. More elaborate details can be found in the full version of [Ant+14].

Job Completion Event This is the easiest event type, capturing the time when the current job i is fully processed. As long as no other event occurs, the work of job i increases at a constant rate p'_i, which can be computed by Equation 5.5. Thus, if the work of i processed at the current time τ is $p_i(\tau)$, job i needs time $\Delta\tau = \frac{p_i - p_i(\tau)}{p'_i}$ to be fully processed.

Simple Rate Change Event A simple rate change event occurs when the right side of the last execution interval of i reaches the release time of some job. Using Lemma 5.20(d), we can see that this only happens when the rate $x'_{r,q,i}$ changes from ${}^1/_{d_i}$ to zero. The corresponding time computes easily as $\Delta\tau = \frac{\hat{r}_{q_{r,i(\tau)+1}}(\tau) - x_{r,q_{r,i(\tau)},i}(\tau)}{{}^1/_{d_i}}$.

Speed Change Event This case is a bit more tedious than the previous one. For each execution interval of each job, we have to compute the job's dual function value at the interval borders, their distance to the next speed thresholds, and the (constant) rate at which these values change. All these values can be easily computed using Lemma 5.20.

Affection Change Event The exact conditions for an affection change event are given in Lemma 5.22. Similar to the previous cases, all that needs to be done is to compute the time when one of the corresponding conditions occurs. More exactly, note that (a) and (b) of Lemma 5.22 correspond to a removal of an edge, whereas (c) corresponds to an addition of an edge. For the removal, we have to compute at which (constant) rates the borders of any execution intervals change and check when they become equal (i.e., the corresponding interval vanishes). For the addition, we have to compute at which (constant) speed the dual lines of two jobs, one in the affection tree and one not) approach each other at the upper envelope.

5.6.2 Putting it All Together

We are now ready to prove the correctness of the algorithm. Note that we handle termination in Theorem 5.24, where we prove a polynomial running time for our algorithm.

Theorem 5.23. *Assuming that the algorithm from Listing 5.1 terminates, it computes an optimal schedule.*

Proof. The algorithm outputs a line schedule S, so by Lemma 5.4, S is optimal if for all jobs j the schedule does exactly p_j work on j. We now show that this is indeed the case.

For a fixed iteration i, we argue that a change in the rate at which work is increasing for j (i.e., a change in p_j') may occur only when an event occurs. This follows from Equation (5.5), since the rate only changes when there is a change in the rate at which the endpoints of intervals move, when there is a change in the speed levels employed in each interval, or when there is an affection change (and hence a change in the intervals of a job or a change in α_j'). These correspond to the events we have defined, which are correctly computed (see Section 5.6.1). Thus, the algorithm recalculates the rates at any event, and by Lemma 5.21 it calculates the correct rates such that $p_j'(\tau) = 0$ for $j \in [i-1]$ and for every τ. This is done until some τ_0 such that $p_i(\tau_0) = p_i$, which corresponds to the job completion event computed by the algorithm (see Section 5.6.1). Thus, we get the invariant that after iteration i we have a line schedule for the first i jobs that does p_j work for every job $j \in [i]$. The theorem follows. □

5.7 The Running Time

The purpose of this section is to prove the following theorem.

Theorem 5.24. *The algorithm from Listing 5.1 takes* $\mathrm{O}(n^4 k)$ *time.*

We do this by upper bounding the number of events that can occur, see Lemma 5.26. This is relatively straightforward for job completion, simple rate change, and speed change events, which can occur $\mathrm{O}(n)$, $\mathrm{O}(n^2)$, and $\mathrm{O}(n^2 k)$ times, respectively. However, bounding the number of times an affection change event can occur is more involved: We show in Lemma 5.25 that whenever an edge is removed from the affection tree, there exists an edge which will never again be in the affection tree. This implies that the total number of affection change events is upper bounded by $O(n^2)$ as well. We then show in Lemma 5.27 that the next event can always be calculated in $O(n^2)$ time, and in Lemma 5.28 that the affection tree can be updated in $O(n)$ time after each affection change event. By combining these results it follows that our algorithm has a running time of $O(n^4 k)$.

Lemma 5.25. *Consider some time τ_0 where an edge (j, j') is removed from the affection tree. Then there exists some edge (u, v) that is also being removed at τ_0 such that (u, v) will not be present for all remaining iterations of the algorithm.*

Proof. First note that by the definition of the affection tree, it must be that the affection $j \to j'$ is being removed. Since j is a parent of j', by Lemma 5.17 we have $d_j < d_{j'}$. Also, by Lemma 5.22, this edge can be removed because either the intersection between j and j' becomes equal to r_j or two intersection points become equal. We handle these cases separately.

In the first case assume that the intersection between j and j' becomes equal to r_j. We show that the affection edge $j \to j'$ cannot be present again. To do this, we show that the invariant of j' not being processed on the (left) upper envelope to the right of r_j is always maintained. This implies that the edge $j \to j'$ is never present again. It is clearly true for τ_0 (say, in iteration i). Assume that for some iteration $i' \geq i$ this invariant is true. If j' is not being raised the invariant will remain true since curves are only raised and not lowered. If j' is being raised, since it is not the lowest density job it must intersect some lower density job j'' (its parent) that is also being raised. Further, since the invariant is true to this point, the intersection is not to the right of r_j. However, while j' is being raised, by Observation 5.19 the intersection between j' and j'' moves only to the left. Since $d_{j''} < d_{j'}$, j' will not be on the upper envelope or left upper envelope to the right of this intersection and the result follows.

In the second case, assume that the intersection between j_1 and j_2 becomes equal to the intersection between j_2 and j_3 and assume, without loss of generality, that $d_{j_1} < d_{j_2} < d_{j_3}$. This implies that the edges (j_1, j_2) and (j_2, j_3) will be removed. We show that the edge (j_2, j_3) will not be present again. First note that $r_{j_2} < r_{j_3}$ since otherwise j_2 would not be processed anywhere, contradicting that the rates at which we raise curves are work-preserving. Similar to the previous argument, we show that j_2 will not be processed on the upper envelope or left upper envelope to the right of r_{j_3} again. This is clearly true at τ_0 (say, in iteration i). Assume for some iteration $i' \geq i$ this invariant is true. Again, if j_2 is not being raised the invariant remains true. If j_2 is being raised, it must intersect a lower density job (its parent) to the left of r_{j_3}. Since this intersection point will move only to the left the result follows. □

Lemma 5.26. *The total number of events throughout the execution of the algorithm is* $O(n^2k)$.

Proof. To prove this we show that the number of events is $O(n^2k)$ for each single type.

- **Job completion event:** Consider a job completion event. Since this is the last event in every iteration, it will occur exactly n times.
- **Simple rate change event:** This occurs during iteration i when the last interval of job i changes from ending before some job's release to at this release. Since we never lower job i's dual line, the rightmost endpoint will only move to the right and therefore will hit each job's release point at most once. So, the total number of simple rate change events is $O(n)$ per iteration, and therefore $O(n^2)$ in total.
- **Affection change event:** These events happen when an edge is either added or removed in the tree. Note that the number of edge additions is bounded by 2 times the number of edge removals, so it suffices to bound the number of removals. By Lemma 5.25, for each (possibly temporarily) removed edge at least one edge is removed permanently. Thus, the total number of such events is $O(n^2)$.
- **Speed change event:** These events occur when the right or left endpoint of an interval for a job j cross a speed threshold. Note that as long as an interval is never removed, each endpoint can only cross each speed threshold once, since by Observation 5.19 the endpoints of intervals only move to the left and dual lines never decrease. If an interval is removed, it remains to verify that it cannot reappear later with either endpoint at a lower speed. An interval is removed when the right and left endpoint become equal on the upper envelope at some point t (by Lemma 5.22). At

this point there must be some job j' of lower density which also intersects at t. However, by Observation 5.19, the left endpoint of j' (or another job of lower density than j' if j''s interval disappears) will only move to the left while this interval is not present. Therefore, even if the interval does reappear, the left and right endpoints will not be at lower speeds. Since each job has at most n intervals and each such interval can cause at most $2k$ speed change events, the total number of speed change events is $\mathrm{O}(n^2 k)$. □

Lemma 5.27. *Calculating the next event takes* $\mathrm{O}(n^2)$ *time.*

Proof. We start by noting that the total number of different intervals during the execution of the algorithm is $\mathrm{O}(n)$. This follows by the fact that a new interval can only be introduced when a new job is released, or a job completes its execution.

To calculate the next event, we calculate when the next event of each type will happen. Then we simply choose the event of the type that will happen sooner. Therefore for the rest we give bounds on the time required to calculate the next event of each type (see also subroutines in Section 5.6.1).

- **Affection change event:** By the observation on the number of intervals, $\mathrm{O}(n)$ time suffices to calculate the next event of this type if it is a removal. On the other hand, if the next event of this type is an addition, $\mathrm{O}(n^2)$-time is required.
- **Speed change event:** For any fixed interval the next speed change event can be calculated in constant time. Therefore, by the observation on the number of intervals, we have that the next such event over all jobs can be computed in time $\mathrm{O}(n)$,
- **Simple rate change event:** $\mathrm{O}(n)$-time is sufficient in order to identify $q_{r,i}$ and $\hat{r}_{q_{r,i}+1}$, and therefore also to calculate this type of event as well.
- **Job completion event:** We have to calculate $y_{r,j}, y_{\ell,j}$ for each of the $\mathrm{O}(n)$ intervals, identify i' and calculate $\delta_{i,i'}$. Therefore, we can calculate the next job completion event in time $\mathrm{O}(n)$.

In total we can calculate the next event in $\mathrm{O}(n^2)$ time. □

Lemma 5.28. *Updating the affection tree takes* $\mathrm{O}(n)$ *time.*

Proof. A simple way to update the affection tree is by recomputing it from scratch at each update. By Lemma 5.11, jobs in the tree always have a higher density than their parents. By Observations 5.7 and 5.8, if a job j is on the upper envelope (or left upper envelope) at some time t and has release time before t, and $j' \neq j$ is the highest-density job on the upper envelope (left upper envelope) at time t, then $j \rightarrow j'$, and for any other job $j'' \neq j'$ of higher density than j on the upper envelope (left upper envelope), $j \nrightarrow j''$. Therefore, for any job j, its children in the affection tree are those highest-density jobs that intersect it on the left endpoint of any of its intervals that begin after j's release. Thus, to compute the affection tree, we can iterate through each interval I of job i that begins afters its release, add as i's children the highest dense jobs that intersect it at I's left endpoint, and recursively do the same for i's children. By the observation that there are at most $2n$ intervals, this takes at most $\mathrm{O}(n)$ time. □

5.8 Conclusion & Outlook

In contrast to previous chapters, which studied speed scaling together with deadline constraints, the present chapter considered a speed scaling problem with a flow time (a.k.a. response time) objective. This is both one of the most well studied scheduling objectives and one of the most important quality of service measures. Intuitively, the response time plus energy objective allows for a user-defined trade-off between energy and responsiveness: by adjusting the energy unit in this linear combination, the user specifies how much energy she is willing to spend on decreasing the response time by one time unit. Grasping this problem's offline complexity is one of the fundamental open problems for speed scaling. This chapter presented an efficient solution for a variant that relaxed this problem in two ways: we considered *fractional* flow and the case of discrete speed levels. The hope is that these relaxations might help to solve the original problem for arbitrary work shops with respect to average response time plus energy. To this end, it seems promising to elaborate on how the presented results and insights might help to solve other relaxations of the original problem.

An obvious open problem is how to resolve the remaining issues in [Col+12] to get an optimal polynomial-time algorithm for the continuous speed setting. Apart from the direct approaches suggested in [Col+12], the results of this chapter may shed some light on this issue. An interesting observation is that [Col+12] gives a simple method to check for optimality, while a corresponding algorithm for our discrete speed setting seems to be more complicated; in particular, not much easier than computing an optimal schedule. At first glance, this difficulty for discrete speeds comes from the fact that an optimal schedule's representation is not necessarily unique (in contrast to the continuous case).

Another unsolved problem is how to extend this chapter's results for discrete speeds to non-fractional response time. Even finding an algorithm that is polynomial in the number of jobs and exponential in the number of speeds would be of both theoretical and practical interest.

CHAPTER 6

Sharing Scalable Resources

"The waste of plenty is the resource of scarcity."

Thomas Love Peacock

THE last part of this thesis introduces a new processor scheduling problem, which is only remotely connected to the speed scaling idea from previous chapters[1]. The proposed scheduling problem is motivated by the observation that, sometimes, it is not a device's speed or energy consumption that limits the progress of a given computation, but the fact that data cannot be provided at the necessary rate. In extreme cases, this may lead to situations where changing the available I/O rate (or bandwidth) by some factor x may directly affect the running time by (approximately) the same factor.

At first glance, this seems more a network issue than a problem of interest for processor scheduling. After all, bandwidth bottlenecks are typically imposed by the interconnection of devices (e.g., networks or data buses), and there is a huge body of literature concerned with such issues on the network layer. However, the analysis in this area typically concentrates on the *network's* performance. In contrast, the model proposed in this chapter focuses on how the distribution of the bandwidth shared by a fixed set of processing units might

[1]See the discussion in Section 6.2.1.

affect their *computational* performance. That is, given some information about the bandwidth requirement of a program (e.g., when does it need how much bandwidth to progress at full speed), the scheduler can speed up critical jobs by a suitable assignment of the available bandwidth to the different processors.

A First Glimpse at the Model From a more abstract point of view, the aforementioned bandwidth scheduling can be seen as a variant of resource constrained scheduling, the bandwidth being an example for the resource. Imagine a system consisting of several identical processors that run at a fixed speed and share a given resource. It is assumed that the resource is the system's performance bottleneck, in the sense that the running time of programs (tasks) depends directly (that is to say, linearly) on the share of the resource they are allowed to use. Each task provides information about its resource requirements by stating what share of the resource it needs at different phases of its processing to run at full speed. Thus, we can imagine a task i to consist of a number n_i of jobs that must be processed sequentially, one after another. Each job represents a phase of the task's processing where the resource requirement is constant. When provided a portion $x \in [0,1]$ of its requested resource share, a job's processing time (i.e., the length of the task's corresponding phase) increases by a factor of $1/x$. We use the term ResourceScaling to refer to this problem. A more formal description of this model is provided in Section 6.2.

We will see, partly in the discussion of related literature in Section 6.1, that this type of resource assignment problem is comparatively complex. Most work considering similar problems seems to be of heuristic nature and analytical results are scarce (and, if surfaced, quite negative). Since we are interested in more analytical insights, this chapter approaches the problem by concentrating on the resource assignment, removing the (classical) scheduling aspect almost completely. That is to say, we will consider a scenario where there is exactly one task for each processor, and each task consists of jobs of unit workload (but different resource requirements). Moreover, we assume discrete time steps (think of processor cycles), such that the scheduler can change the resource assignment only at the beginning of such a time step. As we will see, even this simple setting proves to be challenging.

Chapter Basis The model, analysis, and results presented in the remainder of this chapter are based on the following technical report. An extended version of these results is currently in preparation for submission.

> **2014** (with F. Meyer auf der Heide, L. Nagel, S. Riechers and T. Süß). "Sharing Scalable Resources". In preparation, cf. [Kli+14].

Chapter Outline Section 6.1 surveys related work and describes our contribution in view of known results in this area. A formal model description of the ResourceScaling problem is provided in Section 6.2, which also introduces a graphical representation used for the analysis in later chapters. Section 6.3 provides the reader with some preliminaries for the analysis and discusses a first, simple result for a round robin algorithm. In Section 6.4, we show that the ResourceScaling problem is NP-hard in the number of processors, even in the case of unit size jobs. The main part of this chapter can be found in Section 6.5, where we prove that so called balanced schedules yield a $2 - 1/m$ approximation for ResourceScaling with unit size jobs. We also propose a natural greedy algorithm for computing balanced schedules and show that the proven approximation ratio is tight for this algorithm. As usual, we conclude the chapter with a short résumé and outlook in Section 6.6.

6.1 Related Work & Contribution

Compared to the speed scaling problems considered in previous chapters, the proposed ResourceScaling problem has a quite different background. In essence, it is a classical resource constrained scheduling problem. That is, the scheduler not only has to manage the computational resources (e.g., the assignment of jobs to processors) but also the allocation of one or more additional resources to the currently processed jobs. In our context (processor scheduling), the most obvious examples for such resources are probably bandwidth and memory. However, note that models similar to ours are also used in project planning or for manufacturing systems.

The following discussion focuses on results for so called *discrete-continuous* models, where the computational resource is discrete (e.g., a certain number of processors) and the additional resources are continuous (e.g., bandwidth can be allotted in a continuous manner between the available processors). For a more general overview of resource constrained scheduling, the interested reader is referred to [Leu04, Chapters 23 and 24] and [BEP07, Chapter 12].

Discrete-continuous Scheduling The notion of *discrete-continuous* scheduling traces back to several papers by Józefowska and Węglarz, first and foremost [JW98]. While most results in this area study scenarios where the amount of allocated resources influences the processing time or release dates of jobs (see [JJL07] for a survey on this), Józefowska and Węglarz [JW98] consider the case where the amount of allocated resources influences the processing *speed* of jobs. More exactly, if the function $R_j := \mathbb{R}_{\geq 0} \to [0,1]$ models the share of the resource that job j gets assigned at some time $t \in \mathbb{R}_{\geq 0}$, its workload is processed at a speed of $f_j(R_j(t))$. Here, f_j models how a job's processing speed is affected by the received resource amount and is assumed to be continuous and non-decreasing with $f_j(0) = 0$. Using this resource model, the authors consider the problem of scheduling n non-preemptable (independent) jobs on m processors. They propose an analysis framework based on a mathematical programming formulation and demonstrate it for the objective of minimizing the schedule's makespan. For certain classes of f_j (e.g., convex functions) this yields a simple analytical solution, but finding an optimal solution for more realistic cases (especially concave functions) remains infeasible. The results in [JW98] initiated several research efforts in this area, including a transfer of the methodology to other scheduling variants (e.g., average flow time instead of makespan [JW96]) as well as several heuristic approaches to obtain practical solutions in the general case [Józ+00; Józ+02; Kis05; Wal11]. A detailed and current survey about these results can be found in [Węg+11] (especially Section 7).

Our ResourceScaling problem shares several characteristics with discrete-continuous scheduling problems. In particular, the jobs' resource requirements can be modeled via functions f_j of the form $f_j(R) = \min({}^{R}/_{r_j}, 1)$, where the value r_j denotes the resource requirement of job j (cf. Section 6.2). That is, the speed used to process a job depends linearly on the share of the resource it receives, but is capped at one. Our model contains several other important differences, the most obvious being that the assignment of jobs to processors as well as the order of jobs on a given processor is fixed. While this severely limits the possibilities of the scheduler (which can no longer try to simply distribute the jobs more or less evenly among the available processors), it also allows us to focus on the inherent problem complexity of assigning the continuous resource such that the schedule's makespan is minimized. Note that most of the aforementioned results for the discrete-continuous setting are of heuristic nature and do not provide any provable quality guarantees with respect to the resulting schedules. The cases that can be analyzed analytically turn out to feature very simple optimal solution structures [JW98; Józ+99] (as, for example, processing only one job at a time). In contrast, solution structures in the ResourceScaling problem turn out to be much more diverse and complex. Still, focusing solely on how the resource assignment interacts with a schedule's structure allows us to derive an algorithm with a provably good quality guaranty.

Contribution This chapter introduces a new resource constrained scheduling model for multiple processors, where job processing speeds depend on the assigned share of a common resource. We concentrate on a variant with unit size jobs where the scheduler only has to manage the distribution of the resource among all processors. The objective is to minimize the total makespan (maximum completion time over all jobs). Even for this simple variant, we can prove NP-hardness with respect to the number of processors. While we cannot determine the exact complexity for a constant number of three or more processors, we provide an approximation algorithm that achieves a worst-case approximation ratio of exactly $2 - 1/m$. To the best of our knowledge, this is the first strong analytical result for this type of problem. Our approach uses a hypergraph representation that allows us to capture non-trivial structural properties.

6.2 Model & Notation

We start by defining the model for the general version of the ResourceScaling problem, which considers jobs of arbitrary sizes. Afterward, we discuss an alternative interpretation of our model and its relation to speed scaling. Note that, while the model description considers jobs of arbitrary sizes, the analysis part of this chapter focuses on the case where all jobs have unit size.

6.2.1 Formal Model Description

Consider a system of m identical fixed-speed processors sharing a common resource. At any time step $t \in \mathbb{N}$, the scheduler can distribute the resource between the m processors. To this end, each processor i is assigned a share $R_i(t) \in [0,1]$ of the resource, which it is allowed to use in time step t. It is the responsibility of the scheduler to ensure that the resource is not overused. That is, it must guarantee that $\sum_{i=1}^{m} R_i(t) \leq 1$ holds for all $t \in \mathbb{N}$. For each processor i, there is a sequence of $n_i \in \mathbb{N}$ jobs that must be processed by the processor in the given order. We write (i,j) to refer to the j-th job on processor i. Parallelism is not allowed, such that a processor can process at most one job during any given time step. Each job (i,j) has a *processing volume* (size) $p_{ij} > 0$ and a *resource requirement* $r_{ij} \in [0,1]$. The resource requirement specifies what portion of the resource is needed to process one unit of the job's processing volume in one time step. In general, when a job is granted an x-portion of its resource requirement ($x \in [0,1]$), exactly x units of its processing volume are processed in that time step. There is no benefit in granting a job more than its requested share of the resource. That is, a job's processing cannot be sped up by granting it, for example, twice its resource requirement.

A feasible schedule for an instance of the ResourceScaling problem consists of m resource assignment functions $R_i \colon \mathbb{N} \to [0,1]$ that specify the resource's distribution among the processors for all time steps without overusing the resource. We measure a schedule's quality via its makespan (i.e., the time when all jobs are finished). Our goal is to find a feasible schedule having minimal makespan. To simplify notation, we will often identify a schedule S with its makespan (e.g., by writing S/OPT to denote the makespan of schedule S divided by the makespan of an optimal schedule OPT).

Relation to Speed Scaling An alternative interpretation of our scheduling problem can be obtained by the following observation: Consider a job (i,j) whose processing is started at time step t_1. It receives a share $R_i(t_1) \in [0,1]$ of the resource. By the previous model definition, exactly $\min({R_i(t_1)}/{r_{ij}}, 1)$ units of its processing volume are processed. Similarly, in the next time step $\min({R_i(t_1+1)}/{r_{ij}}, 1)$ units of its processing volume are processed. Consequently, the job is finished at the minimal time step $t_2 \geq t_1$ with $\sum_{t=t_1}^{t_2} \min({R_i(t)}/{r_{ij}}, 1) \geq$

p_{ij} or, equivalently if $r_{ij} > 0$, at the minimal time step $t_2 \geq t_1$ with

$$\sum_{t=t_1}^{t_2} \min(R_i(t), r_{ij}) \geq r_{ij} p_{ij} =: \tilde{p}_{ij}. \tag{6.1}$$

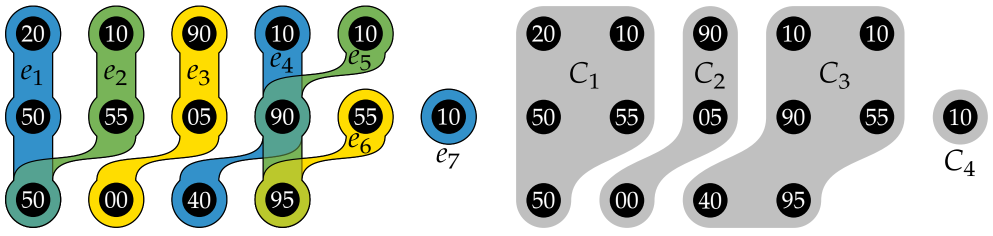
This observation allows us to get rid of the resource aspect and instead think of a job (i, j) to have size $\tilde{p}_{ij}$ and of a processor i to be speed-scalable with $R_i(t)$ denoting the speed it is set to during time step t. The scheduler is in control of the processors' speeds, but it must ensure that the aggregated speed of all processors does never exceed one. Moreover, in addition to the system's maximum speed limit, each job (i, j) is annotated with the maximum speed r_{ij} it can utilize. In this light, our ResourceScaling problem becomes a speed scaling problem to minimize the makespan. Instead of the typical energy constraint, the scheduler is limited by both the system's maximum aggregated speed and a per-job speed limit. Note that the unit size restriction for the ResourceScaling problem translates into the restriction that job sizes $\tilde{p}_{ij}$ equal the corresponding resource requirements r_{ij}. In other words, all jobs must be processable in one time step if run at maximum speed.

During the analysis, it will sometimes be more convenient to think of our problem in the way described above. For example, since the total workload $\sum_{i=1}^{m} \sum_{j=1}^{n_i} \tilde{p}_{ij}$ is processed at a maximum speed of one in any time step, this view on the problem immediately yields the following simple but useful observation:

Observation 6.1. Any feasible schedule needs at least $\sum_{i=1}^{m} \sum_{j=1}^{n_i} r_{ij} p_{ij}$ time steps to finish a given set of jobs with resource requirements r_{ij} and sizes p_{ij}.

Sometimes we will also use the notion *remaining resource requirement* to denote the remaining work of a job's initial workload $\tilde{p}_{ij}$.

Additional Notation & Notions The following additional notions and notation will turn out to be helpful in the analysis and discussion. For a processor i with n_i jobs, we define $n_i(t)$ as the number of unfinished jobs at the start of time step t. In particular, we have $n_i(1) = n_i$. A processor i is said to be *active* at time step t if $n_i(t) > 0$. Similarly, we say that job (i, j) is *active* at time step t if $n_i - n_i(t) = j - 1$ (i.e., if processor i has finished exactly $j - 1$ jobs at the start of time step t). We use $M_j := \{ i \mid n_i \geq j \}$ to denote the set of all



(a) Scheduling graph H_S trying to greedily finish as many jobs as possible.

(b) The connected components of the scheduling graph, ordered from left to right.

Figure 6.1: Hypergraph representation of a schedule for three processors. Resource requirements are given as node labels (in percent). The nodes are laid out such that each row corresponds to the job sequence of one processor (from left to right). The edges correspond to the schedule that prioritizes jobs in order of increasing remaining resource requirement (cf. "Relation to Speed Scaling" in Section 6.2).

processors having at least j jobs to process. Finally, we define $n := \max_i n_i$ as the maximum number of jobs any processor has to process.

6.2.2 Graphical Representation

Given a problem instance of ResourceScaling with unit size jobs and a corresponding schedule S, we can define a weighted hypergraph $H_S = (V, E)$ as follows: The nodes of H_S and their weights correspond to the jobs and their resource requirements, respectively. That is, the node set is given by $V = \{\, (i,j) \mid i = 1,2,\dots,m \land j = 1,2,\dots,n_i \,\}$, and the weight of a node $(i,j) \in V$ is r_{ij}. The edges of H_S correspond to the schedule's time steps and contain the currently active jobs. More formally, the edge $e_t \subseteq V$ for time step t is defined as $e_t := \{\, (i,j) \mid n_i(t) > 0 \land j = n_i - n_i(t) + 1 \,\}$. Thus, if we abuse S to also denote the makespan of schedule S, the edge set of H_S can be written as $E = \{\, e_1, e_2, \dots, e_S \,\}$. We call H_S the *scheduling graph* of S. See Figure 6.1a for an illustration.

Connected Components In Section 6.3.1 and during the analysis in Section 6.5, we will see that the connected components formed by the edges of a scheduling graph H_S carry a lot of structural information about the schedule. To make use of this information, let us introduce some notation that allows us to directly argue via such components. We start with an observation that follows from the construction of H_S.

Observation 6.2. Consider a connected component $C \subseteq V$ of H_S and two time steps $t_1 \leq t_2$ with $e_{t_1} \cup e_{t_2} \subseteq C$. Then, for all $t \in \{ t_1, t_1 + 1, \dots, t_2 \}$, we have $e_t \subseteq C$.

Let N denote the total number of connected components and let C_k denote the k-th connected component (for $k \in \{ 1, 2, \dots, N \}$). Moreover, we use $\#_k$ to denote the number of edges of the k-th component. That is, we have $\#_k = |\{ e_t \in E \mid e_t \subseteq C_k \}|$. Observation 6.2 implies that a component C_k consists of $\#_k$ consecutive time steps. This allows us to order the components such that, for any two components k, k' and edges $e_t \subseteq C_k, e_{t'} \subseteq C_{k'}$ with $t \leq t'$, we have $k \leq k'$. That is, we can think of the components being processed by the processors from left to right. See Figure 6.1b for an illustration.

The maximal size of an edge in the k-th component, which equals the size of its first edge, gives us a rough estimate for the amount of potential parallelism available during the corresponding time steps. Note that while the size of edges e_t is monotonously decreasing in t, a schedule that tries to balance the number of remaining jobs on each processor will decrease the edge size only near the end of a component. We will make use of this fact in the proof of Lemma 6.14. For now, let us honor its foreshadowed importance by the following definition:

Definition 6.3 (Component Class)**.** Given a component C_k, we define its *class* q_k as the size of its first edge. That is, $q_k := |e_t|$ with $t = \min \{ t' \mid e_{t'} \subseteq C_k \}$.

Besides being an upper bound on the size of a component's edges, the class q_k is also decreasing in k. Moreover, Lemma 6.10 will show that a component's class allows us to formulate an important relation between its size and the total number of its edges.

6.3 Preliminaries

This section is intended to make the reader more comfortable with the introduced terms and notions and to equip her with the tools needed for the analysis in later sections. We start by discussing and proving some basic structural properties. Afterward, we analyze a simple round robin algorithm.

6.3.1 Structural Properties

In the remainder of this section, we will use the introduced notions and notation to point out some structural properties of schedules for the ResourceScaling problem with unit size jobs. We start by defining two basic properties any reasonable schedule should have (and show in Lemma 6.6 that this is indeed the case).

Definition 6.4 (Non-wasting Schedule)**.** We call a schedule *non-wasting* if it finishes all active jobs during any time step t with $\sum_{i=1}^{m} R_i(t) < 1$.

Definition 6.5 (Progressive Schedule)**.** We call a schedule *progressive* if at most one job is only partially processed during any time step t. More formally, we require that $|\{\, i \mid n_i(t) = n_i(t+1) \wedge R_i(t) > 0 \,\}| \leq 1$ holds for all $t \in \mathbb{N}$.

Lemma 6.6. *Given an arbitrary schedule S, we can transform it into a non-wasting and progressive schedule S' with $S' \leq S$ (S and S' denoting the corresponding makespans). Moreover, the resulting schedule S' finishes at least one job per time step.*

Proof. Making a given schedule non-wasting is trivial, as given a time step t with $\sum_{i=1}^{m} R_i(t) < 1$ and an active job (i', j'), we can increase $R_{i'}(t)$ until either the job is finished or $\sum_{i=1}^{m} R_i(t) = 1$. In both cases, the schedule's makespan does not increase.

Given a non-wasting schedule S that is not progressive, consider two jobs (i_1, j_1) and (i_2, j_2) on different processors at a time step t such that $n_{i_1}(t) = n_{i_1}(t+1)$, $n_{i_2}(t) = n_{i_2}(t+1)$, and $R_{i_1}(t), R_{i_2}(t) > 0$. We will define the new schedule S' by providing two new resource assignment functions R'_{i_1} and R'_{i_2} that swap some of the resources assigned to jobs (i_1, j_1) and (i_2, j_2). To this end, let $t_1, t_2 > t$ be the time steps in which (i_1, j_1) and (i_2, j_2) are finished, respectively. Without loss of generality, assume $t_1 \leq t_2$. Let $R := \sum_{t'=t+1}^{t_1} R_{i_1}(t')$ denote the total resource assignment that job (i_1, j_1) receives after time step t. If $R \leq R_{i_2}(t)$, we define $R'_{i_1}(t) := R_{i_1}(t) + R$ and $R'_{i_2}(t) := R_{i_2}(t) - R$. This will finish job (i_1, j_1) at time step t, so that the resources formerly assigned to (i_1, j_1) after time step t can be freed. The inequality $t_1 \leq t_2$ implies that job (i_2, j_2) is active during the time steps $t+1$ to t_1. Thus, we can set $R'_{i_1}(t') := 0$ and $R'_{i_2}(t') := R'_{i_2}(t') + R'_{i_1}(t')$ for all $t' \in \{\, t+1, t+2, \ldots, t_1 \,\}$. For all remaining time steps t', we set $R'_{i_1}(t) := R_{i_1}(t)$ and $R'_{i_2}(t) := R_{i_2}(t)$. These changes result in a feasible schedule, do not increase the schedule's makespan, and do not waste any resources. Thus, by iterating this procedure we get a non-wasting and progressive schedule. □

Lemma 6.6 allows us to narrow our study to the subclass of non-wasting and progressive schedules, and from now on we will assume any schedule to have these properties (if not stated otherwise).

Intuitively, good schedules should try to balance the number of remaining jobs on each processor. This may provide the scheduler with more choices to prevent the underutilization of the resource later on (e.g., when only one processor with many jobs of low resource requirements remains). The better part of Section 6.5 serves the purpose of confirming this intuition. In the following, we formalize this balance property and, subsequently, work out further formal and concise properties of balanced schedules.

Definition 6.7 (Balanced Schedule)**.** We call a schedule *balanced* if, whenever a processor i finishes a job at some time step t, any processor i' with $n_{i'}(t) > n_i(t)$ does also finish a job.

Proposition 6.8. *Any balanced schedule features the following properties:*

(a) For all i_1, i_2 with $n_{i_1} \geq n_{i_2}$ and for all $t \in \mathbb{N}$, we have $n_{i_1}(t) \geq n_{i_2}(t) - 1$.

(b) For all i_1, i_2 with $n_{i_1} > n_{i_2}$ and for all $t \in \mathbb{N}$, we have $n_{i_1}(t) \leq n_{i_2}(t) + n_{i_1} - n_{i_2}$.

Proof. Both statements follow easily from the definition of balanced schedules. To see this, first note that both properties hold for $t = 1$, since $n_i(1) = n_i$ for all processors i. Moreover, at any time step t, the number $n_i(t)$ of remaining jobs cannot increase, and does decrease by at most one during the current time step. Thus, it is sufficient to show that if one of the statements holds at some time step t with equality, it still holds at time step $t + 1$. For statement (a), $n_{i_1}(t) = n_{i_2}(t) - 1$ and the balance property imply that if i_1 finishes its job, then so must i_2. Thus, we have $n_{i_1}(t+1) \geq n_{i_2}(t+1) - 1$. The very same argument works for statement (b). □

Proposition 6.9. *Consider a balanced schedule and the set M_j of processors having at least j jobs. Let (i, j) be a job that is active at time step t and assume $n_i(t) > 1$ (i.e., it is not the last job on processor i). Then all processors $i' \in M_j$ are active at time step t.*

Proof. Let $i' \in M_j$ be a processor with at least j jobs and consider the case $n_{i'} \geq n_i$. By Proposition 6.8(a), we have $n_{i'}(t) \geq n_i(t) - 1 > 0$, so processor i is active at time t. If $n_{i'} < n_i$, we can apply Proposition 6.8(b) and get

$$n_{i'}(t) \geq n_{i'} - (n_i - n_i(t)) = n_{i'} - (j - 1) \geq 1. \tag{6.2}$$

The equality uses the fact that job (i, j) is active at time step t, implying that the number $n_i - n_i(t)$ of jobs finished by processor i before time step t is exactly $j - 1$. The last inequality comes from $i' \in M_j$. □

The final structural property of balanced schedules addresses, as indicated earlier, how a component's class allows us to relate its size (number of nodes) to the total number of its edges.

Lemma 6.10. *Consider a non-wasting, progressive, and balanced schedule. The number of nodes and edges in a component are related via the following properties:*

(a) The inequality $|C_k| \geq \#_k + q_k - 1$ holds for all $k \in \{1, 2, \dots, N - 1\}$.

(b) The last component satisfies $|C_N| \geq \#_N$.

Proof. The second statement follows immediately from Lemma 6.6, which states that in each time step (i.e., for each edge) at least one job is finished.

For the first statement, fix a $k \in \{1, 2, \dots, N - 1\}$ and consider the first edge e_t of the component C_k. By definition, this edge consists of q_k different nodes. We now show that each of the remaining $\#_k - 1$ edges adds at least one new node to the component. So fix an edge $e_{t'} \subseteq C_k$ with $t' > t$ and consider the time step $t' - 1$. Since we know that at least one job is finished in every time step (Lemma 6.6) and that S is balanced, at least one of the processors having the maximal number of remaining jobs finishes its current job. More formally, there is some processor $i' = \arg\max_i n_i(t' - 1)$ that finishes its currently active job at time step $t' - 1$. Because of $k \neq N$, we also know that $n_{i'}(t' - 1) > 1$, such that there is a new active job for processor i' at time step t'. This yields the lemma's first statement. □

6.3.2 Warm-up: Approximating via Round Robin

Consider the following simple round robin algorithm for the ResourceScaling problem with unit size jobs: Given a problem instance where the maximal number of jobs on a processor is n, the algorithm operates in n phases. During phase j, it processes the j-th job on each processor, assigning the resource in an arbitrary way to any processors that have not yet finished their j-th job. Note that this algorithm may waste resources (although only between two phases) and is possibly non-progressive. Still, the following theorem shows that it results in schedules that are not too bad.

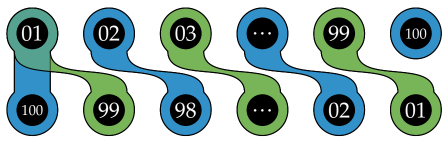
Theorem 6.11. *The round robin algorithm for the ResourceScaling problem with unit job sizes has a worst-case approximation ratio of exactly* 2.

Proof. We start with the upper bound on the approximation ratio. The round robin algorithm needs exactly $\lceil \sum_{i \in M_j} r_{ij} \rceil$ time steps to finish the j-th phase (cf. "Relation to Speed Scaling" in Section 6.2). Thus, the makespan of a round robin schedule can be bounded by

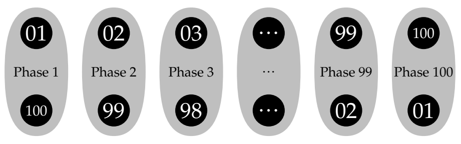
$$\sum_{j=1}^{n} \left\lceil \sum_{i \in M_j} r_{ij} \right\rceil \leq n + \sum_{j=1}^{n} \sum_{i \in M_j} r_{ij}. \tag{6.3}$$

Since any processor can finish at most one job per time step, even an optimal schedule has a makespan of at least n. Observation 6.1 yields another lower bound on the optimal makespan, namely $\sum_{j=1}^{n} \sum_{i \in M_j} r_{ij}$. Together, we get that round robin computes a 2-approximation.

For the lower bound on the approximation ratio, consider the following ResourceScaling problem instance with unit size jobs on two processors: Let $\varepsilon > 0$ and define the resource requirements for the first processor as $r_{1j} := j \cdot \varepsilon$ for $j \in \{1, 2, \ldots, 1/\varepsilon\}$. For the second processor, we define $r_{2j} := (1 + \varepsilon) - r_{1j}$. Note that each processor has to process $n = 1/\varepsilon$ jobs. Figure 6.2 illustrates the instance as well as the resulting optimal and round robin schedules for $\varepsilon = 0.01$. An optimal schedule, shown in Figure 6.2a, will waste no resource at all. In contrast, the round robin schedule, as indicated in Figure 6.2b, wastes a share of $1 - \varepsilon$ of the resource in every second time step. As a result, it needs $2n$ time steps, while the optimal schedule can finish the same workload in $n + 1$ time steps. Thus, for $\varepsilon \to 0$ we get a 2-approximation. □



(a) Optimal schedule, wastes no resources and needs exactly $n + 1$ time steps.



(b) Round robin, two time steps per phase and wastes 99% of the resource at the end of each phase.

Figure 6.2: Worst-case example for round robin schedule.

6.4 Problem Complexity

One of our first major results is the following theorem, showing that the ResourceScaling problem is (even in the case of unit size jobs) NP-hard in the number of processors.

Theorem 6.12. *The ResourceScaling problem with unit size jobs is NP-hard in the number of processors.*

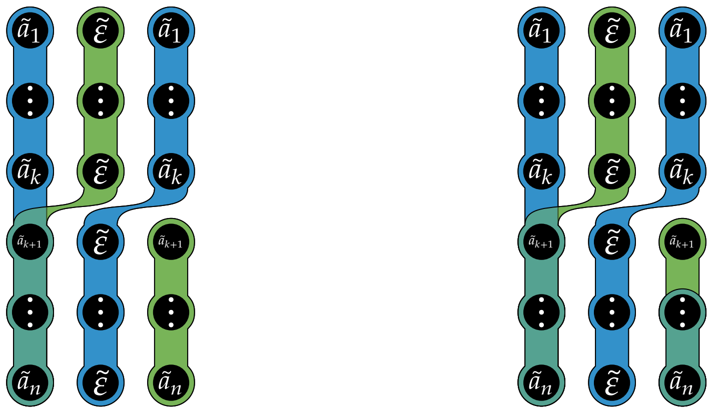
Proof. In the following, we prove the NP-hardness of the ResourceScaling problem with unit size jobs via a reduction from the Partition problem. Our reduction transforms a Partition instance of n elements into a ResourceScaling instance on n processors, each having three jobs to process.

Let $a_1, a_2, \dots, a_n \in \mathbb{N}$ and $S \in \mathbb{N}$ with $\sum_{i=1}^{n} a_i = 2S$ be the input of the Partition instance (w.l.o.g., $S \geq 2$). For our transformation, let $\varepsilon \in (0, {}^1/_n)$ and set $\delta := n\varepsilon < 1$. We define the first and last job on any processor i to have resource requirements $r_{i1} = r_{i3} = \tilde{a}_i := \frac{a_i}{S+\delta}$. The second job on any processor i has a resource requirement of $r_{i2} = \tilde{\varepsilon} := \frac{\varepsilon}{S+\delta}$. Note that the total resource requirement of all jobs is

$$\sum_{i=1}^{n} \frac{a_i + \varepsilon + a_i}{S + \delta} = \frac{4S + \delta}{S + \delta} > 3. \tag{6.4}$$

The last inequality follows from $\varepsilon < {}^1/_n \leq {}^S/_{2n}$ and the definition of δ. By applying Observation 6.1, Equation (6.4) yields that any schedule needs at least four time steps to finish all jobs. To finish our reduction, we show that there is an optimal schedule with makespan 4 if and only if the given Partition instance is a YES-instance (i.e., if it can be partitioned into two sets that sum up to exactly S).

Let us first assume we have a YES-instance of Partition. Assume, without



(a) Optimal schedule for a YES-instance.   (b) Optimal schedule for a NO-instance.

Figure 6.3: Problem instance and schedules used for the reduction from Partition to ResourceScaling with unit size jobs.

loss of generality, that the first k elements form the first partition. One can easily check that the schedule of makespan 4 shown in Figure 6.3a is feasible.

Now assume that we are given a NO-instance of Partition and an optimal schedule for the corresponding instance of ResourceScaling. Without loss of generality, exactly the first k processors finish their jobs in the first time step. This implies $\sum_{i=1}^{k} \tilde{a}_i \leq 1$, which yields the inequality $\sum_{i=1}^{k} a_i \leq S + \delta < S + 1$. Since the given Partition instance is a NO-instance, we also have $\sum_{i=1}^{k} a_i \neq S$. Together this implies $\sum_{i=1}^{k} a_i \leq S - 1$, which, in turn, yields $\sum_{i=k+1}^{n} a_i \geq S + 1$. After the first time step, we have not yet finished the jobs $(k+1, 1), (k+2, 1), \ldots, (n, 1)$. Thus, we need at least two more time steps until we can start working on the jobs $(k+1, 3), (k+2, 3), \ldots, (n, 3)$. The total resource requirement of these jobs is at least

$$\sum_{i=k+1}^{n} \tilde{a}_i = \frac{\sum_{i=k+1}^{n} a_i}{S + \delta} \geq \frac{S+1}{S+\delta} > 1. \tag{6.5}$$

Thus, after the first three time steps, we need at least two more time steps to finish the remaining jobs, yielding a makespan of at least five. □

6.5 Analysis of Balanced Schedules

This section builds up to our second major result, an approximation algorithm with a tight approximation ratio of $2 - {}^1\!/\!_m$. We start by providing two lower bounds for optimal schedules in terms of a given non-wasting or balanced schedule. Afterward, we will use these bounds in the proof of our main result in Theorem 6.15.

6.5.1 Lower Bounds for Optimal Schedules

The following lemma derives a lower bound for OPT by exploiting the fact that within a component any non-wasting schedule always makes full use of the resource; the only situation when the resource may not be fully utilized is during the last time step (edge) of a component.

Lemma 6.13. *Let OPT denote the minimal makespan of a given problem instance and consider the scheduling graph H_S of a non-wasting schedule S. Then OPT can be bounded by*

$$OPT \geq \sum_{k=1}^{N} (\#_k - 1). \tag{6.6}$$

Proof. From Observation 6.1, we immediately get that OPT $\geq \sum_{i=1}^{m} \sum_{j=1}^{n_i} r_{ij}$. Consider a connected component C_k of our schedule containing the edges $t_1, t_1+1, \dots, t_2$. Since S is non-wasting, we must have $\sum_{i=1}^{m} R_i(t) = 1$ for all time steps $t \in \{ t_1, t_1+1, \dots, t_2 - 1 \}$. If there were such a t with $\sum_{i=1}^{m} R_i(t) < 1$, the non-wasting property would imply that all active jobs are finished. But then the edge e_{t+1} would not be part of C_k, yielding a contradiction. For the last time step t_2 of C_k we have $\sum_{i=1}^{m} R_i(t_2) \geq 0$. Moreover, S is feasible and, without loss of generality, does not use more of the resource than necessary. Thus, if T denotes the total length of schedule S, we must have $\sum_{t=1}^{T} \sum_{i=1}^{m} R_i(t) = \sum_{i=1}^{m} \sum_{j=1}^{n_i} r_{ij}$. If we use $e^{(k)}$ to denote the last edge of component C_k, we can bound OPT as follows:

$$\begin{aligned} \text{OPT} &\geq \sum_{i=1}^{m} \sum_{j=1}^{n_i} r_{ij} = \sum_{t=1}^{T} \sum_{i=1}^{m} R_i(t) = \sum_{k=1}^{N} \sum_{e_t \subseteq C_k} \sum_{(i,j) \in e_t} R_i(t) \\ &\geq \sum_{k=1}^{N} \sum_{\substack{e_t \subseteq C_k \\ e_t \neq e^{(k)}}} 1 = \sum_{k=1}^{N} (\#_k - 1). \end{aligned}$$

□

The second lower bound on an optimal schedule's makespan centers around utilizing parallelism. It is obvious that no schedule can have a makespan lower than n, the maximal number of jobs on any one processor. Stated differently, assuming a problem instance where each processor has exactly n jobs, the maximum parallelism that any schedule can exploit in each time step is m. On the other hand, given a schedule with components C_k of class q_k, the maximum parallelism that can be exploited while working on component C_k is q_k. In a sense, the following lemma shows that in the case of balanced schedules this is actually not too bad.

Lemma 6.14. *Let OPT denote the minimal makespan of a given problem instance and remember that n denotes the maximum number of jobs any processor has to process. Given a balanced schedule S and its scheduling graph, OPT and n can be bounded by the inequalities*

$$OPT \geq n \geq \sum_{k=1}^{N-1} \frac{|C_k|}{q_k} + \frac{|C_N|}{m}. \tag{6.7}$$

Proof. Remember that M_j is the set of processors having at least j jobs to process. Since any schedule can process at most one job per processor in every time step, even an optimal schedule needs at least n time steps to finish all jobs. We can write n as $\sum_{(i,j)\in V} {}^{1}/_{|M_j|}$, yielding

$$\begin{aligned} \text{OPT} \geq n &= \sum_{(i,j)\in V} \frac{1}{|M_j|} = \sum_{k=1}^{N} \sum_{(i,j)\in C_k} \frac{1}{|M_j|} \\ &\geq \sum_{k=1}^{N-1} \sum_{(i,j)\in C_k} \frac{1}{|M_j|} + \sum_{(i,j)\in C_N} \frac{1}{m} = \sum_{k=1}^{N-1} \sum_{(i,j)\in C_k} \frac{1}{|M_j|} + \frac{|C_N|}{m}. \end{aligned} \tag{6.8}$$

It remains to show that, for all but the last component, we have

$$\sum_{(i,j)\in C_k} \frac{1}{|M_j|} \geq \frac{|C_k|}{q_k}. \tag{6.9}$$

So fix $k \in \{1, 2, \dots N-1\}$ and let $(i_0, j_0) \in C_k$ be a job of the k-th component with minimal j_0. Furthermore, let t_0 be the first time step when (i_0, j_0) is active. The minimality of j_0 implies that e_{t_0} is the first edge of C_k and, thus, $q_k = |e_{t_0}|$. We distinguish the following cases:

Case 1: $n_{i_0}(t_0) > 1$

By applying Proposition 6.9, we get that all processors $i \in M_{j_0}$ are active at time step t_0. This yields $|M_{j_0}| \leq |e_{t_0}| = q_k$. Moreover, for a job $(i, j) \in C_k$, the minimality of j_0 gives us $|M_{j_0}| \geq |M_j|$. Combining both inequalities implies $|M_j| \leq q_k$. Applying this to the first part of Equation (6.9) eventually yields the desired inequality.

Case 2: $n_{i_0}(t_0) = 1$

In this case, (i_0, j_0) is the last job on processor i_0 at time step t_0. However, for any job $(i, j) \in C_k \setminus e_{t_0}$ we have $n_i(t_0) > 1$. Given such a job, let (i, j') be the job processed on i at time step t_0. Note that we have $j' < j$ and, thus, $M_j \subseteq M_{j'}$. By applying Proposition 6.9, we get that all $i' \in M_{j'}$ are active at time step t_0. Together with $M_j \subseteq M_{j'}$, this yields $|M_j| \leq q_k$. Thus, to prove Equation (6.9), it only remains to show $\sum_{(i,j)\in e_{t_0}} 1/|M_j| \geq \sum_{(i,j)\in e_{t_0}} 1/q_k (= 1)$.

To this end, note that, since C_k is not the last component, there exists at least one job $(i_1, j_1) \in e_{t_0}$ with $n_{i_1}(t_0) > 1$. Let this job be such that j_1 is minimal. Once more, by applying Proposition 6.9 we get that all $i \in M_{j_1}$ are active at time step t_0. Consider a job $(i, j) \in e_{t_0}$ with $i \in M_{j_1}$. If it is the last job on i (i.e., if $n_i(t_0) = 1$), we have $j = n_i$. Together with the definition of M_{j_1} we get $j = n_i \geq j_1$, yielding $|M_j| \leq |M_{j_1}|$. Similarly, if it is not the last job on i (i.e., if $n_i(t_0) > 1$), the minimality of j_1 gives us $|M_j| \leq |M_{j_1}|$. This yields the desired inequality as follows:

$$\sum_{(i,j)\in e_{t_0}} \frac{1}{|M_j|} \geq \sum_{\substack{(i,j)\in e_{t_0} \\ i\in M_{j_1}}} \frac{1}{|M_j|} \geq \sum_{\substack{(i,j)\in e_{t_0} \\ i\in M_{j_1}}} \frac{1}{|M_{j_1}|} = 1. \qquad \square$$

6.5.2 Deriving a $(2 - 1/m)$-Approximation

Finally, we have all the ingredients to prove our main result: an approximation ratio of at most $2 - 1/m$ for any balanced schedule.

Theorem 6.15. *Consider an arbitrary* ResourceScaling *instance with unit size jobs and a feasible schedule S for it that is non-wasting, progressive, and balanced. Then S is a $2 - 1/m$-approximation with respect to the optimal makespan.*

Proof. In the following, let $\#_\emptyset := \sum_{k=1}^{N} \#_k / N$ denote the average number of edges in a component. Our proof uses two bounds on the approximation ratio. The first one follows easily from Lemma 6.13 and leads to a better approximation for instances with small $\#_\emptyset$. The second bound is much more involved and mainly based on Lemma 6.14. It yields a better approximation for instances with large $\#_\emptyset$. To get the first bound, we simply apply Lemma 6.13 and get

$$\frac{S}{\mathrm{OPT}} \le \frac{\sum_{k=1}^{N} \#_k}{\sum_{k=1}^{N} (\#_k - 1)} = \frac{\#_\emptyset}{\#_\emptyset - 1}. \tag{6.10}$$

Let us now consider the second bound, based on Lemma 6.14. Our goal is to show that the inequality

$$\frac{S}{\mathrm{OPT}} \le \frac{m \cdot \#_\emptyset}{\#_\emptyset + m - 1} \tag{6.11}$$

holds. Once this is proven, we can combine both bounds by realizing that the bound from Equation (6.10) is monotonously decreasing in $\#_\emptyset$ and the bound from Equation (6.11) is monotonously increasing in $\#_\emptyset$. Equalizing yields that their minimum's maximum is obtained at $\#_\emptyset = \frac{2m-1}{m-1}$, which results in an approximation ratio of $2 - 1/m$.

The rest of this proof is geared towards proving Equation (6.11). We distinguish two cases. The first case covers the easier part, where we have $\mathrm{OPT} \ge n + 1$. That is, even an optimal solution cannot finish the jobs in n time steps. The second case, where we have $\mathrm{OPT} = n$, turns out to be more difficult to prove. While we can apply a similar analysis, we have to take more care when bounding our algorithm's progress in the first two time steps. Let us start with the analysis of the easier case.

Case 1: $\mathrm{OPT} \ge n + 1$

Applying Lemma 6.14 to this case yields

$$\begin{aligned}\frac{S}{\mathrm{OPT}} &\le \frac{\sum_{k=1}^{N} \#_k}{\sum_{k=1}^{N-1} \frac{|C_k|}{q_k} + \frac{|C_N|}{m} + 1} \le \frac{N \cdot \#_\emptyset}{\sum_{k=1}^{N-1} \frac{\#_k + q_k - 1}{q_k} + \frac{\#_N + m - 1}{m}} \\ &\le \frac{N \cdot \#_\emptyset}{\sum_{k=1}^{N} \frac{\#_k + m - 1}{m}} \le \frac{m \cdot \#_\emptyset}{\#_\emptyset + m - 1}.\end{aligned} \tag{6.12}$$

Case 2: OPT $= n$

If we apply the same analysis as in the first case, we will fall short of our desired approximation ratio. Surprisingly, it turns out to be sufficient to bound only the first two time steps more carefully. The idea of the following analysis is to consider the first two time steps of S and the remaining part of S separately. To this end, first note that we can assume, without loss of generality, that $\#_1 > 1$ (that is, the first two time steps are part of the same component). If this is not the case, our algorithm finishes all active jobs in the first time step and, thus, behaves optimally[2]. Consider the remaining jobs/workloads after the first two time steps. We can regard this as a subinstance of our original problem instance. Let S' denote the subschedule that results from restricting S to time steps $t \geq 3$. We use N', $\#'_k$, q'_k, and n' to refer to the corresponding properties of its scheduling graph $H_{S'}$. Note that we have $N' \geq N - 1$ (because of our assumption $\#_1 > 1$) as well as $N' \cdot \#'_\emptyset = N \cdot \#_\emptyset - 2$ (since exactly two time steps are missing in the subschedule). Moreover, we also have $n' = n - 2$. The inequality $n' \geq n - 2$ is obvious. For $n' \leq n - 2$, note that OPT must finish the jobs in the set $\{\, (i,1) \mid n_i(1) \geq n-1 \,\} \cup \{\, (i,2) \mid n_i(1) \geq n \,\}$ during the first two time steps. Thus, the total resource requirement of these jobs is at most two. Since S is balanced, it will prioritize and, thus, finish these jobs in the first two time steps. This yields $n' \leq n - 2$.

Finally, we can bound our approximation ratio as follows (the first inequality applies Lemma 6.14 to S'):

$$\begin{aligned}
\frac{S}{\text{OPT}} = \frac{N \cdot \#_\emptyset}{2 + n'} &\leq \frac{N \cdot \#_\emptyset}{2 + \sum_{k=1}^{N'-1} \frac{|C'_k|}{q'_k} + \frac{|C'_{N'}|}{m}} \\
&\leq \frac{N \cdot \#_\emptyset}{1 + \frac{1}{m} + \sum_{k=1}^{N'-1} \frac{\#'_k + q'_k - 1}{q'_k} + \frac{\#'_{N'}}{m} + \frac{m-1}{m}} \\
&\leq \frac{N \cdot \#_\emptyset}{1 + \frac{1}{m} + \sum_{k=1}^{N'} \frac{\#'_k + m - 1}{m}} \\
&= \frac{N \cdot m \cdot \#_\emptyset}{m + 1 + N' \cdot \#'_\emptyset + N'(m-1)}
\end{aligned}$$

[2]Formally, this reduces our analysis to a smaller problem instance.

$$\leq \frac{N \cdot m \cdot \#_\emptyset}{2 + (N \cdot \#_\emptyset - 2) + N(m-1)} = \frac{m \cdot \#_\emptyset}{\#_\emptyset + m - 1}.$$

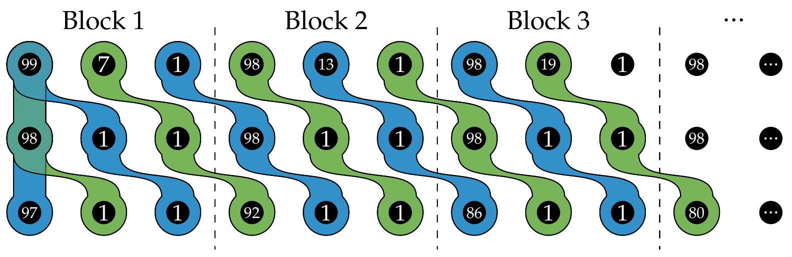
This proves that Equation 6.11 also holds in this case. □

6.5.3 Tight Approximation via a Greedy Algorithm

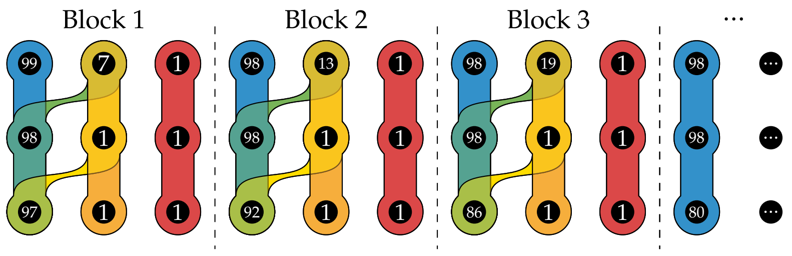
So far, we analyzed the quality of balanced schedules in general but did not yet provide a concrete example of a corresponding algorithm. One of the most natural greedy algorithms schedules jobs by prioritizing processors with a higher number of remaining jobs and, in the case of a tie, by prioritizing jobs with larger remaining resource requirements. We denote this algorithm by the name GreedyBalance. In Section 6.5.2, we saw that balanced schedules and, as a consequence, the greedy algorithm GreedyBalance yield a $2 - 1/m$-approximation for the ResourceScaling problem. In the following, we show that the proven approximation ratio is tight for GreedyBalance.

Theorem 6.16. *The GreedyBalance algorithm for the ResourceScaling problem with jobs of unit size has a worst-case approximation ratio of exactly* $2 - 1/m$.

Proof. Since GreedyBalance computes only balanced schedules, the upper bound follows immediately from Theorem 6.15. For the lower bound, consider a family of problem instances defined as follows: We define blocks of $m \times m$ jobs with resource requirements as described below. For the first block, let $r_{i1} := 1 - i \cdot \varepsilon$ for $i \in \{1, 2, \dots, m\}$, $r_{12} := 1 - \sum_{i=1}^{m}(1 - r_{i1}) + \varepsilon$, and $r_{i2} := \varepsilon$ for $i \in \{2, 3, \dots, m\}$. Moreover, define $r_{ij} := \varepsilon$ for all $i \in \{1, 2, \dots, m\}$ and $j \in \{3, 4, \dots, m\}$. This finishes the first $m \times m$-block of jobs. Having constructed the l-th block, we construct the next block, starting with its first column $j := l \cdot m + 1$. We define $r_{ij} := 1 - (m-1)\varepsilon$ for $i \in \{1, 2, \dots, m-1\}$ and $r_{mj} := 1 - \sum_{i'=1}^{m-1} r_{m-i', j-i'}$. For the second column of this block we set $r_{1,j+1} := 1 - \sum_{i=1}^{m}(1 - r_{ij}) + \varepsilon$, and $r_{i,j+1} := \varepsilon$ for $i \in \{2, 3, \dots, m\}$. To finish the block, we set $r_{ij'} := \varepsilon$ for all $i \in \{1, 2, \dots, m\}$ and $j' \in \{j+2, j+3, \dots, j+m-1\}$. We finish the construction once the next block would contain jobs with negative resource requirements. Note that by choosing ε small enough, we can make this construction arbitrarily long. See Figure 6.4 for an illustration of this construction and the schedules produced by GreedyBalance and an optimal algorithm. Our construction is such that GreedyBalance needs exactly $2m - 1$



(a) An optimal schedule.



(b) Schedule computed by GreedyBalance.

Figure 6.4: Construction and schedules used in the proof of Theorem 6.16 for $m = 3$ and $\varepsilon = 0.01$. Node labels show the corresponding job's resource requirement in percent (e.g., $r_{12} = 0.07$). Note that the optimal schedule needs (essentially) m time steps to finish a block, while S needs $2m - 1$ time steps per block.

time steps per block: By balancing the number of remaining jobs, it is forced to work m time steps on a block's first column (which contains a total resource requirement of roughly m) before it can finish the remaining $m - 1$ columns of a block. In contrast, the optimal algorithm ignores any balancing issues, which eventually allows it to exploit that all diagonals have a total resource requirement of one. □

6.6 Conclusion & Outlook

This chapter introduced a new resource constrained scheduling problem where job processing speeds depend on the share of the resource a job gets assigned. While the problem turned out to be NP-hard in the number of processors (even for unit size jobs), we were able to derive an approximation algorithm with a worst-case approximation ratio of $2 - {}^{1}/_{m}$ (for unit size jobs). Still, the problem's complexity remains unsolved, as we were not yet able to prove or disprove NP-hardness if the number of processors is a constant larger or equal to 3.

The technical report [Kli+14] (in preparation for submission) contains several additional results, as for example a polynomial-time algorithm for the case of two processors. Also, it is not too difficult to show that the ResourceScaling problem is not strongly NP-hard, as there exists a relatively simple (but technical) pseudo-polynomial time algorithm based on a dynamic program.

Besides settling the problem's actual complexity and extending these results to jobs of arbitrary sizes, it seems worthwhile to extend the model to more realistic scenarios. What analytical results are possible if we re-introduce the classical scheduling aspect, where jobs of a task are not a priori fixed to a specific processor? It may also be possible to use our insights to get analytical results in special cases of discrete-continuous models as proposed by Józefowska and Węglarz [JW98].

What might be the most interesting direction, especially given the energy-centric theme of this thesis, are models that consider energy itself as a continuously divisible resource. One might imagine a multiprocessor model in the spirit of the original speed scaling model by Yao et al. [YDS95], but with a shared energy source (similar to what is proposed in [RW11; RW12]).

Bibliography

[AA12] S. Albers and A. Antoniadis. "Race to Idle: New Algorithms for Speed Scaling with a Sleep State". In: *Proceedings of the 23rd Symposium on Discrete Algorithms (SODA)*. 2012.

[AAG11] S. Albers, A. Antoniadis, and G. Greiner. "On Multi-Processor Speed Scaling with Migration". In: *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. San Jose, California, USA: ACM, 2011, pp. 279–288. ISBN: 978-1-4503-0743-7. DOI: `10.1145/1989493.1989539`.

[AF06] S. Albers and H. Fujiwara. "Energy-Efficient Algorithms for Flow Time Minimization". In: *Proceedings of the 23rd Symposium on Theoretical Aspects of Computer Science (STACS)*. Ed. by B. Durand and W. Thomas. Vol. 3884. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, pp. 621–633. DOI: `10.1007/11672142_51`.

[AF07] S. Albers and H. Fujiwara. "Energy-Efficient Algorithms for Flow Time Minimization". In: *ACM Transactions on Algorithms* 3.4 (2007), p. 49. ISSN: 1549-6325.

[AGK12] S. Anand, N. Garg, and A. Kumar. "Resource Augmentation for Weighted Flow-time Explained by Dual Fitting". In: *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Kyoto, Japan: SIAM, 2012, pp. 1228–1241.

[Alb10] S. Albers. "Energy-Efficient Algorithms". In: *Comm. of the ACM* 53.5 (2010), pp. 86–96. DOI: `10.1145/1735223.1735245`.

[Alb11] S. Albers. “Algorithms for Dynamic Speed Scaling”. In: *Proc. of the 28th International Symp. on Theoretical Aspects of Computer Science (STACS)*. Schloss Dagstuhl, 2011, pp. 1–11. DOI: `10.4230/LIPIcs.STACS.2011.1`.

[AMS07] S. Albers, F. Müller, and S. Schmelzer. “Speed Scaling on Parallel Processors”. In: *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. San Diego, California, USA: ACM, 2007, pp. 289–298. ISBN: 978-1-59593-667-7.

[Ant+14] A. Antoniadis, N. Barcelo, M. Consuegra, P. Kling, M. Nugent, K. Pruhs, and M. Scquizzato. “Efficient Computation of Optimal Energy and Fractional Weighted Flow Trade-off Schedules”. In: *Proceedings of the 31st Symposium on Theoretical Aspects of Computer Science (STACS)*. 2014. In press.

[App13a] Apple. *Apple and the Environment*. Dec. 2013. URL: `http://www.apple.com/environment/renewable-energy/`.

[App13b] Apple. *OS X Mavericks - Advanced Technologies*. 2013. URL: `http://www.apple.com/osx/advanced-technologies/`.

[AWT09] L. L. Andrew, A. Wierman, and A. Tang. “Optimal Speed Scaling Under Arbitrary Power Functions”. In: *ACM SIGMETRICS Performance Evaluation Review* 37.2 (Oct. 2009), pp. 39–41. ISSN: 0163-5999. DOI: `10.1145/1639562.1639576`.

[Ban+08a] N. Bansal, D. P. Bunde, H.-L. Chan, and K. Pruhs. “Average Rate Speed Scaling”. In: *Proceedings of the 8th Latin American Conference on Theoretical Informatics (LATIN)*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 240–251. ISBN: 3-540-78772-0, 978-3-540-78772-3.

[Ban+08b] N. Bansal, H.-L. Chan, T.-W. Lam, and L.-K. Lee. “Scheduling for Speed Bounded Processors”. In: *Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP)*. Reykjavik, Iceland: Springer-Verlag, 2008, pp. 409–420. ISBN: 978-3-540-70574-1. DOI: `10.1007/978-3-540-70575-8_34`.

[Ban+09] N. Bansal, H.-L. Chan, K. Pruhs, and D. Katz. “Improved Bounds for Speed Scaling in Devices Obeying the Cube-Root Rule”. In: *Proceedings of the 36th International Colloquium on Automata, Languages and Programming (ICALP)*. Ed. by S. Albers, A. Marchetti-Spaccamela, Y. Matias, S. Nikoletseas, and W. Thomas. Vol. 5555. Springer, 2009, pp. 144–155. ISBN: 978-3-642-02926-4.

[Bap06] P. Baptiste. “Scheduling Unit Tasks to Minimize the Number of Idle Periods: A Polynomial Time Algorithm for Offline Dynamic Power Management”. In: *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithm (SODA)*. ACM, 2006, pp. 364–367. DOI: `10.1145/1109557.1109598`.

[Bar+13] N. Barcelo, D. Cole, D. Letsios, M. Nugent, and K. Pruhs. “Optimal Energy Trade-off Schedules”. In: *Sustainable Computing: Informatics and Systems* 3 (3 2013), pp. 207–217.

[Bar+91] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier, and D. Shasha. “On-line Scheduling in the Presence of Overload”. In: *Proc. of the 32nd Symp. on Foundations of Computer Science (FOCS)*. 1991, pp. 100–110. DOI: `10.1109/SFCS.1991.185354`.

[BC09] N. Bansal and H.-L. Chan. “Weighted Flow Time Does Not Admit O(1)-Competitive Algorithms”. In: *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. New York, New York: Society for Industrial and Applied Mathematics, 2009, pp. 1238–1244.

[BCD07] P. Baptiste, M. Chrobak, and C. Dürr. “Polynomial Time Algorithms for Minimum Energy Scheduling”. In: *Proceedings of the 15th Annual European Symposium on Algorithms (ESA)*. Springer, 2007, pp. 136–150. DOI: `10.1007/978-3-540-75520-3_14`.

[BCP09] N. Bansal, H.-L. Chan, and K. Pruhs. “Speed Scaling with an Arbitrary Power Function”. In: *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. New York, New York: Society for Industrial and Applied Mathematics, 2009, pp. 693–701.

[BCP13] N. Bansal, H.-L. Chan, and K. Pruhs. “Speed Scaling with an Arbitrary Power Function”. In: *ACM Transactions on Algorithms Algorithms (TALG)* 9.2 (Mar. 2013), 18:1–18:14. ISSN: 1549-6325. DOI: `10.1145/2438645.2438650`.

[BE98] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Vol. 53. Cambridge University Press, 1998.

[BEP07] J. Błażewicz, K. H. Ecker, and E. Pesch. *Handbook on Scheduling: From Theory to Applications*. Springer, 2007.

[BG08] B. D. Bingham and M. R. Greenstreet. “Energy Optimal Scheduling on Multiprocessors with Migration”. In: *Proceedings of the 2008 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*. Washington, DC, USA: IEEE Computer

Society, 2008, pp. 153–161. ISBN: 978-0-7695-3471-8. DOI: `10.1109/ISPA.2008.128`.

[BH07] L. A. Barroso and U. Hölzle. "The Case for Energy-Proportional Computing". In: *Computer* 40.12 (2007), pp. 33–37.

[BKP04] N. Bansal, T. Kimbrel, and K. Pruhs. "Dynamic Speed Scaling to Manage Energy and Temperature". In: *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 2004, pp. 520–529. DOI: `10.1109/FOCS.2004.24`.

[BKP07] N. Bansal, T. Kimbrel, and K. Pruhs. "Speed Scaling to Manage Energy and Temperature". In: *Journal of the ACM* 54.1 (2007), pp. 1–39. DOI: `10.1145/1206035.1206038`.

[BN09] N. Buchbinder and J. Naor. *The Design of Competitive Online Algorithms via a Primal-Dual Approach*. Hanover, MA, USA: Now Publishers Inc., 2009. ISBN: 160198216X, 9781601982162. DOI: `10.1561/0400000024`.

[BPS07] N. Bansal, K. Pruhs, and C. Stein. "Speed Scaling for Weighted Flow Time". In: *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Society for Industrial and Applied Mathematics. Philadelphia, USA: Society for Industrial and Applied Mathematics, 2007, pp. 805–813. ISBN: 978-0-898716-24-5.

[BPS09] N. Bansal, K. Pruhs, and C. Stein. "Speed Scaling for Weighted Flow Time". In: *SIAM Journal on Computing* 39.4 (2009), pp. 1294–1308.

[Bro+00] D. M. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P. W. Cook. "Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors". In: *IEEE Micro* 20.6 (2000), pp. 26–44. ISSN: 0272-1732. DOI: `10.1109/40.888701`.

[Bun06] D. P. Bunde. "Power-Aware Scheduling for Makespan and Flow". In: *Proceedings of the 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. Cambridge, Massachusetts, USA: ACM, 2006, pp. 190–196. ISBN: 1-59593-452-9. DOI: `10.1145/1148109.1148140`.

[BV04] S. P. Boyd and L. Vandenberghe. *Convex Optimization*. Seventh. Cambridge University Press, 2004.

[Cha+07] H.-L. Chan, W.-T. Chan, T.-W. Lam, L.-K. Lee, K.-S. Mak, and P. W. H. Wong. "Energy Efficient Online Deadline Scheduling". In: *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. New Orleans, Louisiana: Society for Industrial and Applied Mathematics, 2007, pp. 795–804. ISBN: 978-0-898716-24-5.

[Cha+09] H.-L. Chan, J. W.-T. Chan, T.-W. Lam, L.-K. Lee, K.-S. Mak, and P. W. H. Wong. "Optimizing Throughput and Energy in Online Deadline Scheduling". In: *ACM Transactions on Algorithms* 6.1 (Dec. 2009), 10:1–10:22. ISSN: 1549-6325. DOI: 10.1145/1644015.1644025.

[Che+04] J.-J. Chen, H.-R. Hsu, K.-H. Chuang, C.-L. Yang, A.-C. Pang, and T.-W. Kuo. "Multiprocessor Energy-efficient Scheduling with Task Migration Considerations". In: *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS)*. 2004, pp. 101–108. DOI: 10.1109/EMRTS.2004.1311011.

[CKM12] A. Cord-Landwehr, P. Kling, and F. Mallmann-Trenn. "Slow Down and Sleep for Profit in Online Deadline Scheduling". In: *Proceedings of the 1st Mediterranean Conference on Algorithms (MedAlg)*. Ed. by G. Even and D. Rawitz. Vol. 7659. LNCS. Springer, 2012, pp. 218–231.

[CLL10a] H.-L. Chan, T.-W. Lam, and R. Li. "Tradeoff between Energy and Throughput for Online Deadline Scheduling". In: *Proc. of the 8th Intl. Workshop on Approximation and Online Algorithms (WAOA)*. Springer, 2010, pp. 59–70. DOI: 10.1007/978-3-642-18318-8_6.

[CLL10b] S.-H. Chan, T. W. Lam, and L.-K. Lee. "Non-clairvoyant Speed Scaling for Weighted Flow Time". In: *Proceedings of the 18th annual European Symposium on Algorithms (ESA), Part I*. 2010, pp. 23–35.

[Col+12] D. Cole, D. Letsios, M. Nugent, and K. Pruhs. "Optimal Energy Trade-off Schedules". In: *Proceedings of the 3rd IEEE International Green Computing Conference (IGCC)*. Los Alamitos, CA, USA: IEEE Computer Society, 2012, pp. 1–10. DOI: 10.1109/IGCC.2012.6322257.

[DH14] N. R. Devanur and Z. Huang. "Primal Dual Gives Almost Optimal Energy Efficient Online Algorithms". In: *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2014, pp. 1123–1140.

[Fac13] Facebook. *Open Compute Project*. 2013. URL: http://www.opencompute.org.

[GKP12] A. Gupta, R. Krishnaswamy, and K. Pruhs. "Online Primal-Dual For Non-linear Optimization with Applications to Speed Scaling". In: *Proceedings of the 10th Workshop on Approximation and Online Algorithms (WAOA)*. 2012.

[GNS09] G. Greiner, T. Nonner, and A. Souza. "The Bell is Ringing in Speed-Scaled Multiprocessor Scheduling". In: *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. Calgary, AB, Canada: ACM, 2009, pp. 11–18. ISBN: 978-1-60558-606-9.

[Goo13] Google. *Google Data Centers*. 2013. URL: `http://www.google.com/about/datacenters/`.

[Gra66] R. L. Graham. "Bounds for Certain Multiprocessing Anomalies". In: *Bell System Technical Journal* 45.9 (1966), pp. 1563–1581.

[Han+10] X. Han, T.-W. Lam, L.-K. Lee, I. K. K. To, and P. W. H. Wong. "Deadline Scheduling and Power Management for Speed Bounded Processors". In: *Theoretical Computer Science* 411 (40-42 Sept. 2010), pp. 3587–3600. ISSN: 0304-3975. DOI: `10.1016/j.tcs.2010.05.035`.

[IMP11] S. Im, B. Moseley, and K. Pruhs. "A Tutorial on Amortized Local Competitiveness in Online Scheduling". In: *SIGACT News* 42.2 (2011), pp. 83–97. DOI: `10.1145/1998037.1998058`.

[IP05] S. Irani and K. R. Pruhs. "Algorithmic Problems in Power Management". In: *ACM SIGACT News* 36.2 (2005), pp. 63–76.

[ISG03] S. Irani, S. Shukla, and R. Gupta. "Algorithms for Power Savings". In: *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Baltimore, Maryland: Society for Industrial and Applied Mathematics, 2003, pp. 37–46. ISBN: 0-89871-538-5.

[ISG07] S. Irani, S. Shukla, and R. Gupta. "Algorithms for Power Savings". In: *ACM Transactions on Algorithms* 3.4 (Nov. 2007). ISSN: 1549-6325. DOI: `10.1145/1290672.1290678`.

[JJL07] A. Janiak, W. Janiak, and M. Lichtenstein. "Resource Management in Machine Scheduling Problems: A Survey". In: *Decision Making in Manufacturing and Services* 1.12 (2007), pp. 59–89.

[Józ+00] J. Józefowska, M. Mika, R. Różycki, G. Waligóra, and J. Weglarz. "Solving the Discrete-continuous Project Scheduling Problem via its Discretization". In: *Mathematical Methods of Operations Research* 52.3 (2000), pp. 489–499.

[Józ+02] J. Józefowska, M. Mika, R. Różycki, G. Waligóra, and J. Węglarz. "A Heuristic Approach to Allocating the Continuous Resource in Discrete-continuous Scheduling Problems to Minimize the Makespan". In: *Journal of Scheduling* 5.6 (2002), pp. 487–499.

[Józ+99] J. Józefowska, M. Mika, R. Różycki, G. Waligóra, and J. Wglarz. "Discrete-continuous Scheduling to Minimize the Makespan for Power Processing Rates of Jobs". In: *Discrete Applied Mathematics* 94.1 (1999), pp. 263–285.

[JW96] J. Józefowska and J. Wglarz. "Discrete-continuous Scheduling Problems -- Mean Completion Time Results". In: *European Journal of Operational Research* 94.2 (1996), pp. 302–309.

[JW98] J. Józefowska and J. Węglarz. "On a Methodology for Discrete-continuous Scheduling". In: *European Journal of Operational Research* 107.2 (1998), pp. 338–353.

[Kis05] T. Kis. "A Branch-and-cut Algorithm for Scheduling of Projects with Variable-intensity Activities". In: *Mathematical Programming* 103.3 (2005), pp. 515–539.

[Kli+14] P. Kling, F. Meyer auf der Heide, L. Nagel, S. Riechers, and T. Süß. "Sharing Scalable Resources". 2014. In preparation.

[KP13] P. Kling and P. Pietrzyk. "Profitable Scheduling on Multiple Speed-Scalable Processors". In: *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. New York, NY, USA: ACM, 2013, pp. 251–260. ISBN: 978-1-4503-1572-2. DOI: 10.1145/2486159.2486183.

[Lab+84] J. Labetoulle, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. "Preemptive scheduling of uniform machines subject to release dates". In: *Progress in combinatorial optimization*. Ed. by P. H. R. Academic Press, 1984, pp. 245–261.

[Lam+07] T.-W. Lam, L.-K. Lee, I. K. K. To, and P. W. H. Wong. "Energy Efficient Deadline Scheduling in Two Processor Systems". In: *Proceedings of the 18th International Conference on Algorithms and Computation (ISAAC)*. Sendai, Japan: Springer-Verlag, 2007, pp. 476–487. ISBN: 3-540-77118-2, 978-3-540-77118-0.

[Lam+08] T.-W. Lam, L.-K. Lee, I. K. K. To, and P. W. H. Wong. "Speed Scaling Functions for Flow Time Scheduling Based on Active Job Count". In: *Proceedings of the 16th Annual European Symposium on Algorithms (ESA)*. Ed. by D. Halperin and K. Mehlhorn. Vol. 5193. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, pp. 647–659. DOI: 10.1007/978-3-540-87744-8_54.

[Leu04] J. Y.-T. Leung. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Chapman & Hall/CRC, 2004.

[LY05] M. Li and F. F. Yao. "An Efficient Algorithm for Computing Optimal Discrete Voltage Schedules". In: *SIAM Journal on Computing* 35.3 (Sept. 2005), pp. 658–671. ISSN: 0097-5397. DOI: 10.1137/050629434.

[ML02] J. Markoff and S. Lohr. "Intel's Huge Bet Turns Iffy". In: *The New York Times* (Sept. 2002).

[MV13] N. Megow and J. Verschae. "Dual Techniques for Scheduling on a Machine with Varying Speed". In: *Proceedings of the 40th International Colloquium on Automata, Languages, and Programming (ICALP)*. Riga, Latvia, 2013, pp. 745–756. ISBN: 978-3-642-39205-4. DOI: 10.1007/978-3-642-39206-1_63.

[Ngu13] K. Nguyen. "Lagrangian Duality in Online Scheduling with Resource Augmentation and Speed Scaling". In: *Proceedings of the 21st European Symposium on Algorithms (ESA)*. Ed. by H. Bodlaender and G. Italiano. Vol. 8125. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 755–766. ISBN: 978-3-642-40449-8. DOI: 10.1007/978-3-642-40450-4_64.

[PS10] K. Pruhs and C. Stein. "How to Schedule When You Have to Buy Your Energy". In: *Proc. of the 13th/14th Workshop on Approximation Algorithms for Comb. Optimization Problems/Randomization and Computation (APPROX/RANDOM)*. Springer, 2010, pp. 352–365. DOI: 10.1007/978-3-642-15369-3_27.

[PST04] K. Pruhs, J. Sgall, and E. Torng. "Online Scheduling". In: *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Ed. by J. Y.-T. Leung. Chapman & Hall/CRC, 2004. Chap. 15.

[PUW04] K. Pruhs, P. Uthaisombut, and G. Woeginger. "Getting the Best Response for Your Erg". In: *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory (SWAT)*. Vol. 3111. Springer, 2004, pp. 14–25. DOI: 10.1007/978-3-540-27810-8_3.

[PUW08] K. Pruhs, P. Uthaisombut, and G. Woeginger. "Getting the Best Response for Your Erg". In: *ACM Transactions on Algorithms* 4.3 (July 2008), 38:1–38:17. ISSN: 1549-6325. DOI: 10.1145/1367064.1367078.

[RW11] R. Różycki and J. Węglarz. "Power-aware Acheduling of Preemptable Jobs on Identical Parallel Processors to Minimize Makespan". In: *Annals of Operations Research* (2011), pp. 1–18. ISSN: 0254-5330. DOI: 10.1007/s10479-011-0957-5.

[RW12] R. Różycki and J. Węglarz. “Power-aware Scheduling of Preemptable Jobs on Identical Parallel Processors to Meet Deadlines”. In: *European Journal of Operational Research* 218.1 (2012), pp. 68–75. ISSN: 0377-2217. DOI: `10.1016/j.ejor.2011.10.017`.

[Ste13] S. Stein. *What the next iPhone really needs: Better battery life*. Aug. 5, 2013. URL: `http://reviews.cnet.com/8301-19512_7-57597012-233/what-the-next-iphone-really-needs-better-battery-life/`.

[Vaz01] V. V. Vazirani. *Approximation Algorithms*. New York, NY, USA: Springer-Verlag New York, Inc., 2001. ISBN: 3-540-65367-8.

[Wal11] G. Waligóra. “Heuristic Approaches to Discrete-continuous Project Scheduling Problems to Minimize the Makespan”. In: *Computational Optimization and Applications* 48.2 (2011), pp. 399–421.

[Węg+11] J. Węglarz, J. Józefowska, M. Mika, and G. Waligóra. “Project Scheduling with Finite or Infinite Number of Activity Processing Modes -- A Survey”. In: *European Journal of Operational Research* 208.3 (2011), pp. 177–205.

[YDS95] F. F. Yao, A. J. Demers, and S. Shenker. “A Scheduling Model for Reduced CPU Energy”. In: *Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS)*. 1995, pp. 374–382.