Scheduling Algorithms for Saving Energy and Balancing Load

DISSERTATION

zur Erlangung des akademischen Grades

doctor rerum naturalium
(Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät II
Humboldt Universität zu Berlin

von
Herrn M.Sc. Antonios Antoniadis

Tag der mündlichen Prüfung: 03. August 2012

Abstract

In this thesis we study problems of scheduling tasks in computing environments. We consider both the modern objective function of minimizing energy consumption, and the classical objective of balancing load across machines.

We first investigate offline deadline-based scheduling in the setting of a single variable-speed processor that is equipped with a sleep state. The objective is that of minimizing the total energy consumption. Apart from settling the complexity of the problem by showing its NP-hardness, we provide a lower bound of 2 for general convex power functions, and a particular natural class of schedules called s_{crit}-schedules. We also present an algorithmic framework for designing good approximation algorithms. For general convex power functions our framework improves the best known approximation-factor from 2 to $4/3$. This factor can be reduced even further to $137/117$ for a specific well-motivated class of power functions. Furthermore, we give tight bounds to show that our framework returns optimal s_{crit}-schedules for the two aforementioned power-function classes.

We then focus on the multiprocessor setting where each processor has the ability to vary its speed. Job migration is allowed, and we again consider classical deadline-based scheduling with the objective of energy minimization. We first study the offline problem and show that optimal schedules can be computed efficiently in polynomial time for any convex and non-decreasing power function. Our algorithm relies on repeated maximum flow computations. Regarding the online problem and power functions $P(s) = s^{\alpha}$, where s is the processor speed and $\alpha > 1$ a constant, we extend the two well-known single-processor algorithms *Optimal Available* and *Average Rate*. We prove that *Optimal Available* is α^{α}-competitive as in the single-processor case. For *Average Rate* we show a competitive factor of $(2\alpha)^{\alpha}/2 + 1$, i.e., compared to the single-processor result the competitive factor increases by an additive constant of 1.

With respect to load balancing, we consider offline load balancing on identical machines, with the objective of minimizing the current load, for temporary unit-weight jobs. The problem can be seen as coloring n intervals with k colors, such that for each point on the line, the maximal difference between the number of intervals of any two colors is minimal. We prove that a coloring with maximal difference at most one is always possible, and develop a fast polynomial-time algorithm for generating such a coloring. Regarding the online version of the problem, we show that the maximal difference in the size of color classes can become arbitrary high for any online algorithm. Lastly, we prove that two generalizations of the problem are NP-hard. In the first we generalize from intervals

to d-dimensional boxes while in the second we consider multiple-intervals, i.e., specific subsets of disjoint intervals must receive the same color.

Zusammenfassung

Diese Arbeit beschäftigt sich mit Scheduling von Tasks in Computersystemen. Wir untersuchen sowohl die in neueren Arbeiten betrachtete Zielfunktion zur Energieminimierung als auch die klassische Zielfunktion zur Lastbalancierung auf mehreren Prozessoren.

Beim Speed-Scaling mit Sleep-State darf ein Prozessor, der zu jedem Zeitpunkt seine Geschwindigkeit anpassen kann, auch in einen Schlafmodus bzw. Schlafzustand übergehen. Wir untersuchen Termin-basiertes Speed-Scaling mit Sleep-State. Ziel ist es, den Energieverbrauch zu minimieren. Wir zeigen die NP-Härte des Problems und klären somit den Komplexitätsstatus. Wir beweisen eine untere Schranke für die Approximationsgüte von 2 für eine spezielle natürliche Klasse von Schedules, die wir s_{crit}-Schedules nennen. Das Ergebnis gilt für allgemeine konvexe Funktionen, die den Energieverbrauch spezifizieren. Ferner entwickeln wir eine Familie von Algorithmen, die gute Approximationsfaktoren liefert: Für allgemeine konvexe Funktionen, die den Energieverbrauch spezifizieren, können wir damit den bisher besten bekannten Approximationsfaktor von 2 auf $4/3$ verbessern. Für eine spezielle in der Literatur verbreitete Klasse von Funktionen können wir diesen Faktor noch weiter auf $137/117$ senken. Danach zeigen wir, dass unsere Familie von Algorithmen optimale Lösungen für die Klasse der s_{crit}-Schedules liefert.

Anschließend widmen wir unsere Aufmerksamkeit dem folgenden Termin-basierten Scheduling-Problem. Es seien mehrere Prozessoren gegeben, wobei jeder einzelne Prozessor zu jedem Zeitpunkt seine Geschwindigkeit anpassen kann. Migration von Tasks sei erlaubt. Ziel ist es wie zuvor, den Energieverbrauch des erzeugten Schedules zu minimieren. Für den Offline-Fall entwickeln wir einen Polynomialzeit-Algorithmus, der optimale Schedules für beliebige konvexe Funktionen, die den Energieverbrauch spezifizieren, mittels wiederholter Flusskonstruktionen berechnet. Für das Online-Problem und Funktionen der Form $P(s) = s^\alpha$ erweitern wir die zwei bekannten Ein-Prozessor-Algorithmen *Optimal Available* und *Average Rate*. Hierbei sei s die Prozessorgeschwindigkeit und $\alpha > 1$ eine beliebige Konstante. Wir beweisen, dass *Optimal Available* wie im Ein-Prozessor-Fall α^α-kompetitiv ist. *Average Rate* hat eine Güte von $(2\alpha)^\alpha/2 + 1$. Im Vergleich zum Ein-Prozessor-Fall erhöht sich somit der kompetitive Faktor additiv um die Konstante 1.

Bei der Lastbalancierung auf mehreren Prozessoren betrachten wir Offline-Load-Balancing auf identischen Maschinen. Unser Ziel ist es, die Current-Load für temporäre Tasks mit identischem Gewicht zu minimieren. Diese Problemstellung ist äquivalent zu der folgenden: Gegeben seien n sich teilweise überlappende Intervalle. Diese sind mit k Farben so zu färben, dass zu jedem Punkt für je zwei Farben die Differenz der Anzahlen der Intervalle, die mit diesen zwei Farben gefärbt sind, minimiert wird. Wir zeigen, dass eine Färbung mit maximaler Imbalance von eins immer existiert und entwickeln einen effizienten Algorithmus, der solche Färbungen liefert. Für den Online-Fall des Problems zeigen wir, dass die maximale Imbalance für jeden Algorithmus unbeschränkt groß werden kann. Zum Schluss beweisen wir die NP-Härte von zwei Verallgemeinerungen des Problems. Bei der ersten betrachten wir d-dimensionale Intervalle, bei der zweiten werden mehrere disjunkte Intervalle als zusammengehörig betrachtet und müssen daher dieselbe Farbe erhalten.

To my beloved parents

Acknowledgments

First and foremost I would like to thank my advisor Prof. Susanne Albers for giving me the opportunity to become a member of her group. I feel fortunate to have worked with her and am deeply grateful for her guidance and support.

I would also like to thank the members of the “Algorithms and Complexity” group for interesting discussions and for providing a friendly and stimulating working environment: my room-mate Matthias Hellwig, Chien-Chung Huang, Falk Hüffner, Matthias Killat, Pascal Lenzner, Carsten Moldenhauer, Ralf Oelschlägel, Achim Passen, Eva Sandig and Alexander Souza.

Contents

List of Figures

Chapter 1

Introduction

Scheduling not only forms a large class of optimization problems that have been studied extensively since the 1950's. It also is encountered every day in our lives. A cooking recipe, a bus schedule, or a shift work timetable, all are feasible solutions to particular scheduling problems.

More specifically, any problem of allocating resources over time to a collection of activities, subject to certain constraints, and with the goal of optimizing an objective function can be classified as a scheduling problem. For instance, in the cooking-recipe example, we can view the chefs, the stove, and the cooking devices as resources, and the steps that need to be performed as activities. Naturally, there are certain constraints; some tasks have to be performed before others and not every chef is skilled enough to perform more than one task simultaneously. As an objective function it is sensible to consider that of minimizing the total cooking time.

The problems studied in this thesis are concerned with scheduling in computing environments, where the resources typically are processors, and the activities or jobs correspond to the programs to be executed. There exist many natural objective functions for problems in this setting. Two extensively studied examples are that of minimizing the time at which the last job is completed, and minimizing the response time of the programs executed. We focus on problems with a different classical objective function, namely that of load balancing, as well as problems with the more modern objective of minimizing the energy consumed by the processor.

Due to the fact that energy is a limited and expensive resource, the energy-efficiency of computing environments is increasingly becoming an issue of critical importance. For example, the power consumption of big data centers is nowadays comparable to that of a small city [2]. Saving energy is however also crucial on smaller computing environments - especially on mobile devices where limitations in battery technology play an important role.

To this end, modern microprocessors have various capabilities for energy saving. One of them, *dynamic speed scaling* refers to the ability of a processor to dynamically set the speed/frequency depending on the present workload. High speed implies high performance. On the other hand, the higher the speed, the higher the energy consumption. This behavior can be modeled by means of a power function. The integration of a *sleep-state*, is another common energy-saving technique employed by many contemporary microprocessors. In a deep sleep state, a processor uses negligible or no energy. Transitioning the processor back to the active state, which is necessary in order to execute tasks, usually incurs some fixed amount of energy consumption. The algorithmic challenge in such microprocessor settings is to fully utilize the energy-saving capabilities of the processor while maintaining a quality of service. Scheduling problems with the objective of minimizing energy consumption are classified as *energy-efficient scheduling* problems. For a thorough survey on algorithms for energy-efficient scheduling problems, see [5].

When scheduling in a multi-processor environment, it is often desirable to distribute the load of the jobs to be processed as “evenly” as possible on the processors. Take as an example the following problem (inspired by an example in [54]). We wish to transmit videos over network channels. Some videos are of higher image quality than others and therefore cause a higher load per time unit to the channel that they are assigned to. We would like to assign the videos to channels in a way such that at any point in time, the load assigned to different channels is as balanced as possible. This is crucial in order to provide a high quality of service, i.e. a smooth video transmission. The above example describes a *machine load balancing* setting with the objective of minimizing *current load*. That is we want to keep the load in the channels balanced at any point in time. Another commonly used objective in machine load balancing problems is that of minimizing the *peak load*, i.e., the maximum load over machines and time. The objectives of peak load and current load are quite different. In the context of our video transmission example, minimizing peak load would not make much sense since an extraordinary high load at some point in time would allow us to keep the load unbalanced at later timepoints, resulting in transmissions of worse quality.

More formally, in machine load balancing, each job has a starttime, an endtime and a weight. Throughout the time interval between its starttime and its endtime the job has to be assigned to a unique machine. The load of any machine at a given timepoint is defined as the sum of the weights of the jobs assigned to it at that timepoint. The objective can be that of minimizing either peak load or current load. An interesting subproblem of machine load balancing is that where all jobs have a unit weight. In the context of our example this can be thought of as transmitting only videos of the same image quality.

1.1 Preliminaries

Before presenting our contributions in more detail, we explain how the performance of algorithms is evaluated, and introduce the needed definitions and terminology.

1.1.1 Analysis of Algorithms

It is common practice to analyse and evaluate the performance of an algorithm in terms of its running time, i.e., the number of steps it requires in order to produce the desired result. The running time usually grows with the input size, and moreover can differ for different inputs of the same size. Therefore, the running time of an algorithm is commonly measured by a function of its input size that determines the number of computational steps required for the worst case input of that size.

NP-Hardness and Approximation Algorithms

There exist problems that are solvable in polynomial-time, and problems that provably require time superolynomial in the input size to be solved. A universally accepted distinction is drawn between these two classes of problems: the first are said to be tractable, or easy, whereas the second intractable, or hard (see [32] for an extensive discussion).

Some of the problems considered in this thesis belong to the class of NP-complete problems. Problems in this class are in a way equivalent with respect to tractability: a polynomial-time algorithm for any NP-complete problem would imply tractability for every problem in this class. On the other hand, a proof that there exists such an intractable problem would also prove that no NP-complete problem can be tractable. The question on whether NP-complete problems are tractable or not, is one possible formulation of the infamous open problem $\text{P}\overset{?}{=}\text{NP}$. The class of NP-hard problems contains all problems that are at least as hard as NP-complete problems.

Unless P=NP, which is considered highly unlikely, we cannot hope to design a polynomial-time algorithm that computes an optimal solution for an NP-hard optimization problem. Nevertheless many such problems are too important to be left unaddressed. One way to circumvent NP-hardness is by developing *approximation algorithms*, i.e., algorithms that run in polynomial-time and compute a feasible solution.

All optimization problems studied in this thesis are minimization problems. Definitions throughout this section will be given with this in mind, but the terminology can easily be extended to include maximization problems as well.

We evaluate the quality of the solution returned by an approximation algorithm to a given instance in terms of its *performance ratio.*

Definition. ***([10])*** *For any instance x of a given optimization problem, and any feasible solution y of x, the* **performance ratio** *of y with respect to x is defined as*

$$R(x, y) = \frac{cost(x, y)}{cost^*(x)},$$

where $cost(x, y)$ denotes the cost of solution y to instance x and $cost^*(x)$ denotes the cost of an optimal solution to instance x.

In order to evaluate the performance of an approximation algorithm, we make use of the *approximation ratio/factor*, which, loosely speaking, is the worst-case performance ratio.

Definition. ***([10])*** *We say that an approximation algorithm $\mathcal{A}$ for an optimization problem is an* **r-approximation** *algorithm (or achieves an* **approximation ratio/factor** *of r), if given any input instance x of the problem, the performance ratio of the approximate solution $\mathcal{A}(x)$ that $\mathcal{A}$ outputs on instance x, is bounded by r. That is, for every input x, it holds that,*

$$R(x, \mathcal{A}(x)) \leq r.$$

Note that the value of the approximation ratio is always greater than or equal to 1 (an approximation ratio of 1 is achieved only by algorithms that compute optimal solutions). Some NP-hard optimization problems admit *polynomial-time approximation schemes* which, loosely speaking, means that there exist approximation algorithms for the respective problem with an approximation factor arbitrary close to 1.

Definition. *A* **polynomial-time approximation scheme (PTAS),** *for an optimization problem P, is an algorithm that given as input any instance x of P and a constant $\epsilon > 0$, produces, in time polynomial to the size of x, a feasible solution with a performance ratio at most $1 + \epsilon$.*

Online Algorithms

So far we have limited our discussion to the traditional *offline* setting. That is, we assumed that the algorithm knows the whole input in advance. Often, especially when studying scheduling problems, it is more realistic to consider an *online* setting, i.e., the input becomes available in a piecewise fashion while the algorithm is running.

In order to evaluate the performance of online algorithms, we resort to competitive analysis [50]. Informally, competitive analysis compares the performance of the online algorithm with that of an optimal offline algorithm.

Definition. *We say that an online algorithm $\mathcal{A}$ for an optimization problem is* c**-competitive** *(or achieves a* **competitive ratio/factor** *of* c*), if for any given input instance* x *of the problem,*

$$\frac{cost_{\mathcal{A}}(x)}{cost^*(x)} \leq c$$

holds, where $cost_{\mathcal{A}}(x)$ *denotes the cost of the solution that* $\mathcal{A}$ *returns on* x *and* $cost^*(x)$ *denotes the cost of an optimal offline solution to instance* x*.*

1.1.2 Scheduling

In general, a scheduling problem can be described by:

- the set of resources available,
- the activities (jobs) to which the resources should be allocated over time,
- the constraints involved, and
- the objective function.

Since the number of possible scheduling problems is vast, we only describe the resources, jobs, constraints and objective functions that appear in the problems studied throughout the text.

Resources

For the problems considered here, the resources can always be assumed to be microprocessors. We study both single-processor and multi-processor settings. Furthermore in several problems the processor(s) may have the capability to vary the speed at which jobs are processed, or may be equipped with a sleep state.

Jobs and Constraints

We consider two different models for jobs. In the first, each job is described by a *release time*, that is a timepoint at which it is released and can from now on be processed, a *deadline*, i.e., a timepoint at which the processing of the job has to be completed, and a *processing volume* that represents the amount of processing required for this job. The processing volume for each job can be seen as the number of CPU-cycles required by the job, and has to be finished in the interval between its release time and its deadline. We call this *classical deadline-based scheduling*.

In the second model, each job is described by a *starttime* when we start processing the job, and an *endtime* at which we end the processing of the job. Each job is assumed to be processed during the whole interval defined by its starttime and endtime.

We say that *preemption* is allowed when a processor may pause the execution of a job and continue it later. In the multi-processor setting we may allow job *migration*. This means that a preempted job can continue its processing on a different processor, but at a later timepoint. A schedule subject to the problem constraints is called a *feasible schedule*.

Objective Functions

The scheduling problems in this thesis consider one of the following two objective functions:

- *Energy Minimization:* The goal is to produce a schedule that minimizes the total energy consumption among all feasible schedules.
- *Maximum Load-Imbalance Minimization:* The goal is to minimize the maximum imbalance among the loads assigned to machines at any timepoint. Note that this corresponds to the current load objective.

1.2 Overview

As already mentioned, this thesis studies a number of energy efficient and machine load balancing scheduling problems. The main body of the text is organized in three chapters. We give an overview of our contributions as they are presented in each chapter.

• Race to Idle

In Chapter 2 we investigate the offline energy-conservation problem where a single variable-speed processor is equipped with a sleep state. Executing jobs at high speeds and then setting the processor asleep is an approach that can lead to further energy savings compared to standard dynamic speed scaling. We consider classical deadline-based scheduling, i.e., each job is specified by a release time, a deadline, and a processing volume. Irani et al. [39] devised an offline 2-approximation algorithm for general convex power functions. Their algorithm constructs s_{crit}-schedules, that process all jobs at a speed of at least a “critical speed”. Roughly speaking, the critical speed is the speed that yields the smallest energy consumption while jobs are processed.

First we settle the computational complexity of the optimization problem by proving its NP-hardness. Additionally we develop a lower bound of 2 on the approximation factor that algorithms constructing s_{crit}-schedules can achieve. This lower bound can also be shown to hold for any algorithm that minimizes the energy expended for processing jobs.

We then present an algorithmic framework for designing good approximation algorithms. For general convex power functions, we derive an approximation factor of $4/3$. For power functions of the form $P(s) = \beta s^\alpha + \gamma$, where s is the processor speed, and $\beta, \gamma > 0$ as well as $\alpha > 1$ are constants, we obtain an approximation factor of $137/117 < 1.171$. We conclude the chapter by proving that our framework yields the best possible approximation guarantees for the class of s_{crit}-schedules and the above mentioned classes of power functions. For general convex power functions we give another 2-approximation algorithm and for power functions $P(s) = \beta s^\alpha + \gamma$, we present tight upper and lower bounds on the approximation factor. The factor is exactly $eW_{-1}(-e^{-1-1/e})/(eW_{-1}(-e^{-1-1/e}) + 1) < 1.211$, where W_{-1} is the lower branch of the Lambert W function.

• Multiprocessor Speed Scaling

Chapter 3 is devoted to multi-processor speed scaling. We again consider classical deadline-based scheduling. The differences to the setting in Chapter 2 are: (1) we are given m parallel variable-speed processors instead of a single processor, and (2) none of the m processors is equipped with a sleep state. Furthermore we assume that job migration is allowed, i.e., whenever a job is preempted it may be moved to a different processor.

We first study the offline problem and show that optimal schedules can be computed efficiently in polynomial time given any convex non-decreasing power function. In contrast to a previously known strategy that resorts to linear programming, our algorithm is fully combinatorial and relies on repeated maximum flow computations.

For the online problem, we extend two algorithms *Optimal Available* and *Average Rate* proposed by Yao et al. [55] for the single-processor setting. In this setting, we concentrate on power functions $P(s) = s^\alpha$. We prove that *Optimal Available* is α^α-competitive, as in the single-processor case. For *Average Rate* we show a competitiveness of $(2\alpha)^\alpha/2 + 1$, i.e., compared to the single-processor result the competitive factor increases by an additive constant of 1.

• Load Balancing with Unit-Weight Jobs

Chapter 4 focuses on a machine load balancing problem. More specifically we consider offline load balancing with identical machines and the objective of minimizing the current load where all jobs have unit weights. The problem can be reformulated as follows: we wish to color n intervals with k colors such that at each point on the line, the maximal difference between the number of intervals of any two colors is minimal. In this formulation, every interval models the interval between the starttime and the endtime of a job and every color corresponds to a machine. Additionally, minimizing the maximum imbalance of colors at any timepoint is equivalent to minimizing the current load for unit-weight jobs. As we will see, the studied problem is also closely related to discrepancy theory.

At first, we prove the somewhat surprising fact that a coloring with maximal difference at most one (or equivalently a schedule where the load is ideally balanced at all timepoints) always exists. We then consider the online scenario for the problem where intervals (jobs) arrive over time and the color (machine) has to be decided upon arrival. We show that in this scenario, the maximal difference in the size of color classes can become arbitrarily high for any online algorithm.

Finally we study generalizations of the problem. First, we generalize the problem to d dimensions, i.e., the intervals to be colored are replaced by d-dimensional boxes and show that a solution with imbalance at most one is not always possible. Furthermore we show that for any $d \geq 2$ and $k \geq 2$ it is NP-complete to decide if such a solution exists which implies the NP-hardness of the respective minimization problem. Another interesting generalization of the problem is to consider multiple intervals, i.e., each job is active for a number of disjoint intervals. We show that the problem on multiple intervals is also NP-hard.

Note

The thesis is based on the following publications:

- Susanne Albers and Antonios Antoniadis. Race to idle: new algorithms for speed scaling with a sleep state. In *Proc. 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1266-1285, 2012. (Chapter 2)

- Susanne Albers, Antonios Antoniadis and Gero Greiner. On multi-processor speed scaling with migration. In *Proc. 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 279-288, 2011. (Chapter 3)

- Antonios Antoniadis, Falk Hüffner, Pascal Lenzner, Carsten Moldenhauer and Alexander Souza. Balanced Interval Coloring. In *Proc. 28th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 531-542, 2011. (Chapter 4)

Chapter 2

Race to Idle

Dynamic speed scaling is one of the most common techniques applied to a processor in order to reduce its energy consumption. However, even at low speed levels such a processor consumes a significant amount of static energy, caused e.g. by leakage current. For this reason, and in order to save further energy, modern processors are typically equipped with speed scaling capabilities as well as stand-by or *sleep states*. This combination of speed scaling and low-power states suggests the technique *race-to-idle*: Execute tasks at high speed levels, then transition the processor to a sleep state. This can reduce the overall energy consumption. The race-to-idle concept has been studied in a variety of settings and usually leads to energy-efficient solutions, see e.g. [1, 13, 31, 33, 41].

We adopt a model introduced by Irani et al. [39] to combine speed scaling and power-down mechanisms. The problem is called *speed scaling with sleep state*. Consider a variable-speed processor that, at any time, resides in an *active state* or a *sleep state*. In the active state the processor can execute jobs, where the energy consumption is specified by a general convex, non-decreasing power function P. If the processor runs at speed s, with $s \geq 0$, then the required power is $P(s)$. We assume $P(0) > 0$, i.e. even at speed 0, when no job is processed, a strictly positive power is required. In the active state, energy consumption is power integrated over time. In the sleep state the processor consumes no energy but cannot execute jobs. A *wake-up* operation, transitioning the processor from the sleep state to the active state, requires a fixed amount of $C > 0$ energy units. A power-down operation, transitioning from the active to the sleep state, does not incur any energy.

We consider classical deadline-based scheduling. We are given a sequence $\sigma = J_1, \ldots, J_n$ of n jobs. Each job J_i is specified by a release time r_i, a deadline d_i and a processing volume v_i, $1 \leq i \leq n$. Job J_i can be feasibly scheduled in the interval $[r_i, d_i)$. Again, the processing volume is the amount of work that must be completed on the job. If J_i is processed at constant speed s, then it takes v_i/s time units to finish the job. We may assume that each job is processed at a fixed speed, since by the convexity of the power function P it is not beneficial to process a job at varying speed. Preemption of jobs is allowed, i.e. at any time the processing of a job may be suspended and resumed later. The goal is to construct a feasible schedule minimizing energy consumption.

Given a schedule $\mathcal{S}$, let $E(\mathcal{S})$ denote the energy incurred. This energy consists of two components, the *processing energy* and the *idle energy*. The processing energy $E_p(\mathcal{S})$ is incurred while the processor executes jobs. There holds $E_p(\mathcal{S}) = \sum_{i=1}^{n} v_i P(s_i)/s_i$, where s_i is the speed at which J_i is processed. The idle energy $E_i(\mathcal{S})$ is expended while the processor resides in the active state but does not process jobs and whenever a wake-up operation is performed. We assume that initially, prior to the execution of the first job, the processor is in the sleep state. Suppose that $\mathcal{S}$ contains T time units in which the processor is active but not executing jobs. Let k be the number of wake-up operations. Then, there holds $E_i(\mathcal{S}) = T \cdot P(0) + kC$.

Irani et al. [39] observed that in speed scaling with sleep state there exists a *critical speed* s_{crit}, which is the most efficient speed to process jobs. Speed s_{crit} is the smallest value minimizing $P(s)/s$, and will be important in various algorithms.

Previous Work

Speed scaling and power-down mechanisms have been studied extensively over the past years and we review the most important results relevant to our work. Here, we concentrate on deadline-based scheduling on a single processor. We will cover the multiprocessor case in the introduction of Chapter 3. There exists a considerable body of literature addressing dynamic speed scaling if the processor is *not* equipped with a sleep state. In a seminal paper Yao, Demers and Shenker [55] showed that the offline problem is polynomially solvable. They gave an efficient algorithm, called *YDS* according to the initials of the authors, for constructing minimum energy schedules. Refinements of the algorithm were given in [44–46].

The well-known cube-root rule for CMOS devices states that the speed s of a processor is proportional to the cube-root of the power or, equivalently, that power is proportional to s^3. The algorithms literature considers generalizations of this rule. Early and in fact most of the previous work assumes that if a processor runs at speed s, then the required power is $P(s) = s^\alpha$, where $\alpha > 1$ is a constant. More recent research even allows for general convex non-decreasing power functions $P(s)$. *YDS* was originally presented for power functions $P(s) = s^\alpha$, where $\alpha > 1$, but can be extended to arbitrary convex functions P, see Irani et al. [39]. For power functions $P(s) = s^\alpha$, various online algorithms were presented in [15, 18, 19, 55].

Baptiste [20] studied a problem setting where a *fixed-speed* processor has a sleep state. All jobs must be processed at this fixed speed level, and whenever the processor is in the active state 1 energy unit is consumed per time unit. Using a dynamic programming approach, Baptiste showed that the offline problem of minimizing the number of idle periods in this setting, is polynomially solvable if all jobs have unit processing time. In a subsequent significant paper Baptiste, Chrobak and Dürr [21] used a clever and sophisticated dynamic programming technique to extend Baptiste's approach to arbitrary job processing times. Additionally, they show how to use the dynamic programming table for the above problem in order to minimize the total energy consumption. We will refer to the corresponding polynomial time algorithm as *BCD*.

Irani et al. [39] initiated the study of speed scaling with sleep state. They consider arbitrary convex power functions. For the offline problem they devised a polynomial time 2-approximation algorithm. The algorithm first executes *YDS* and identifies job sets that must be scheduled at speeds higher than s_{crit} according to this policy. All remaining jobs are scheduled at speed s_{crit}. The complexity of the offline problem was unresolved. Irani and Pruhs [38] stated that determining the complexity of speed scaling with sleep state is an intellectually intriguing problem. For the online problem Irani et al. [39] presented a strategy that transforms a competitive algorithm for speed scaling without sleep state into a competitive algorithm for speed scaling with sleep state. For functions $P(s) = s^\alpha + \gamma$, where $\alpha > 1$ and $\gamma > 0$, Han et al. [36] showed an $(\alpha^\alpha + 2)$-competitive algorithm.

Contribution

This chapter investigates the offline setting of speed scaling with sleep state. We consider general convex power functions, which are motivated by current processor architectures and applications, see also [17]. Moreover, we consider the family of functions $P(s) = \beta s^\alpha + \gamma$, where $\alpha > 1$ and $\beta, \gamma > 0$. Speed scaling without sleep state has mostly addressed power functions $P(s) = s^\alpha$. The family $P(s) = \beta s^\alpha + \gamma$ is the natural generalization.

First, in Section 2.1 we develop a complexity result as well as lower bounds. We prove that speed scaling with sleep state is NP-hard and thereby settle the complexity of the offline problem. This hardness result holds even for very simple problem instances consisting of so-called tree-structured jobs and a piecewise linear power function. Hence, interestingly, while the setting with a fixed-speed processor, studied by Baptiste et al. [21], admits polynomial time algorithms, the optimization problem turns NP-hard for a variable-speed processor. As for lower bounds, we refer to a schedule S as an s_{crit}-schedule if every job is processed at a speed of at least s_{crit}. We prove that, for general convex power functions, no algorithm constructing s_{crit}-schedules can achieve an approximation factor smaller than 2. This statement also holds true even for tree-structured jobs and piecewise linear power functions. The lower bound implies that the offline algorithm by Irani et al. [39] attains the best possible approximation ratio among s_{crit}-based algorithms. Furthermore, to obtain smaller approximation factors, one has to use speeds smaller than s_{crit}. Our lower bound construction can be used to show a second result: For general convex power functions, no algorithm minimizing the processing energy of schedules can achieve an approximation factor smaller than 2. Both lower bound statements hold for any algorithm, whose running time might even be exponential.

In Section 2.2 we present a general, generic polynomial time algorithm for speed scaling with sleep state. All three algorithms devised in this chapter are instances of the same algorithmic framework. Our general algorithm combines ***YDS*** and ***BCD***. The main ingredient is a new, specific speed s_0 that determines when to switch from ***YDS*** to ***BCD***. Job sets that must be processed at speeds higher than s_0 are scheduled using ***YDS***. All other jobs are processed at speed s_0. Even though our approach is very natural and simple, it allows us to derive significantly improved approximation factors. For general convex power functions we present a $4/3$-approximation algorithm by choosing s_0 such that $P(s_0)/s_0 = \frac{4}{3}P(s_{crit})/s_{crit}$. The main technical contribution is to properly analyze the algorithmic scheme. The challenging part is to prove that in using speed s_0, but no lower speed levels, we do not generate too much extra idle energy, compared to that of an optimal schedule (cf. Lemmas 3 and 6 for the 4/3-approximation). In Section 2.3 we study power functions $P(s) = \beta s^\alpha + \gamma$ and develop an approximation factor of $137/117 < 1.171$ by setting $s_0 = \frac{117}{137}s_{crit}$.

In Section 2.4 we reconsider s_{crit}-schedules and demonstrate that our algorithmic framework yields the best possible approximation guarantees for the considered power functions. For general convex power functions we give another 2-approximation algorithm, matching our lower bound and the upper bound by Irani et al. [39]. More importantly, we prove tight upper and lower bounds on the best possible approximation ratio achievable for power functions $P(s) = \beta s^\alpha + \gamma$. The ratio is exactly equal to $eW_{-1}(-e^{-1-1/e})/(eW_{-1}(-e^{-1-1/e})+1)$, where W_{-1} is the lower branch of the Lambert W function. The ratio is upper bounded by 1.211.

Summing up, in this chapter we settle the computational complexity of speed scaling with sleep state, provide small constant-factor approximation guarantees for the problem, and settle the performance for the class of s_{crit}-schedules.

2.1 Complexity and lower bounds

In this section we first prove NP-hardness of speed scaling with sleep state. Then we present two lower bounds on the performance of s_{crit}-schedules.

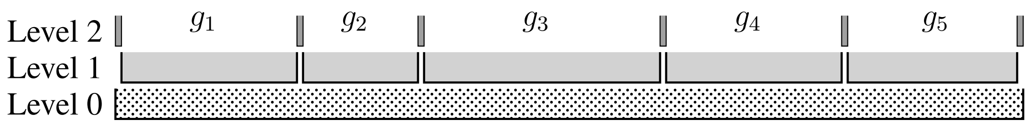


Figure 2.1: The execution intervals of the jobs of an $\mathcal{I}_S$ instance, with $n = 5$.

A problem instance of speed scaling with sleep state is *tree-structured* if, for any two jobs J_i and J_j and associated execution intervals $I_i = [r_i, d_i)$ and $I_j = [r_j, d_j)$, there holds $I_i \subseteq I_j$, $I_j \subseteq I_i$ or $I_i \cap I_j = \emptyset$.

Theorem 1. *Speed scaling with sleep state is NP-hard, even on tree-structured problem instances.*

We proceed to describe a reduction from the NP-complete *Partition* problem [32]. In the *Partition* problem we are given a finite set A of n positive integers $a_1, a_2, \dots a_n$, and the problem is to decide whether there exists a subset $A' \subset A$ such that $\sum_{a_i \in A'} a_i = \sum_{a_i \in A \setminus A'} a_i$. Let a_{max} be the maximal element of A, i.e. $a_{max} = \max_{i \in \{1,\dots n\}} a_i$. We assume $a_{max} \geq 2$ since otherwise the *Partition* problem is trivial to decide.

Let $\mathcal{I}_p$ be any instance of *Partition* with associated set A. The corresponding instance $\mathcal{I}_S$ of speed scaling with sleep state is constructed as follows. For $1 \leq i \leq n$, set $L_i = 2 - \frac{a_{max}-1}{a_{max}^2} a_i$. The job set $\mathcal{J}$ of $\mathcal{I}_S$ can be partitioned into three levels. We first create $n+1$ jobs of level 2, comprising $\mathcal{J}_2 \subset \mathcal{J}$. The i-th job of $\mathcal{J}_2$, with $1 \leq i \leq n+1$, has a release time of $(i-1)\epsilon + \sum_{j=1}^{i-1} L_j$ and a deadline of $i\epsilon + \sum_{j=1}^{i-1} L_j$, where ϵ is an arbitrary positive constant. The processing volume of the i-th job of $\mathcal{J}_2$ is equal to ϵs_{crit}. For level 1, we construct n jobs forming the set $\mathcal{J}_1 \subset \mathcal{J}$. The i-th job of level 1, with $1 \leq i \leq n$, has a release time of $i\epsilon + \sum_{j=1}^{i-1} L_j$, a deadline of $i\epsilon + \sum_{j=1}^{i} L_j$, and a processing volume of $l_i = L_i a_{max} - a_i$. From now on we will also use the term *gap* to refer to the intervals where jobs of $\mathcal{J}_1$ can be executed. More specifically, gap g_i is the interval defined by the release time and the deadline of the i-th job of $\mathcal{J}_1$. Finally, there is only one job J_0 of level 0. It has a release time of 0, a deadline of $(n+1)\epsilon + \sum_{i=1}^{n} L_i$ and a processing volume of $B = \sum_{i=1}^{n} a_i/2$. Figure 2.1 depicts a small example of the above construction.

Note that $\mathcal{I}_S$ is tree-structured. We set the cost of a wake-up operation equal to $C = a_{max}$. The power function is defined as follows:

$$P(s) = \begin{cases} a_{max}, & 0 \leq s \leq a_{max}, \\ \frac{4}{9}s + \frac{5}{9}a_{max}, & a_{max} < s \leq 10a_{max}, \\ 2s - 15a_{max}, & 10a_{max} < s. \end{cases}$$

It is easy to verify that P is convex and continuous, that $s_{crit} = 10a_{max}$, and that $P(s_{crit})/s_{crit} = 1/2$. For the sake of simplicity we assume that the processor is in

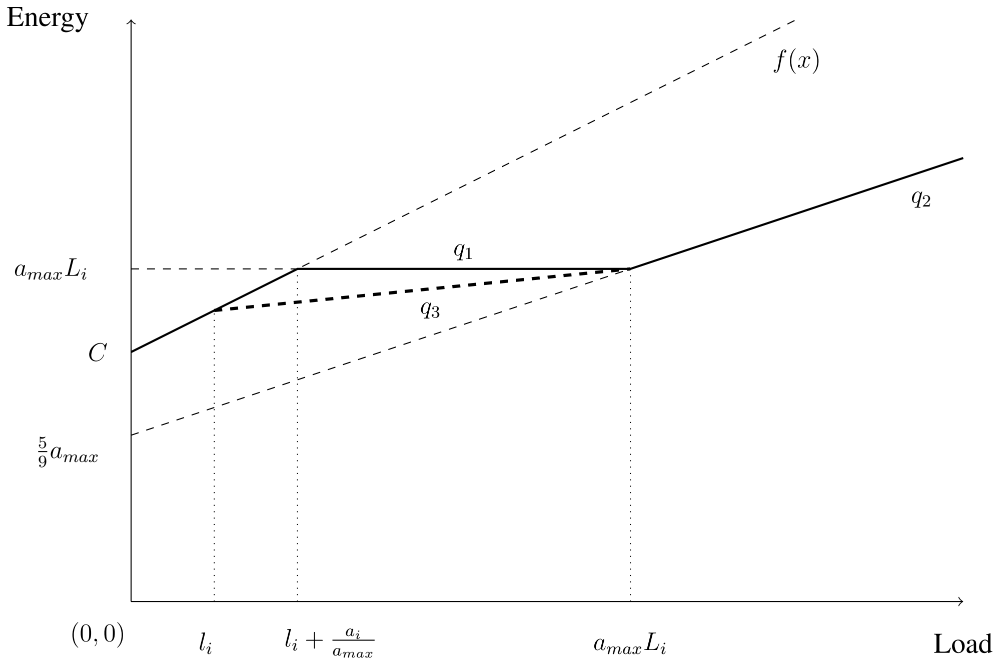


Figure 2.2: Energy consumption in gap g_i as a function of the load executed in g_i.

the active state just before the first job gets executed. This is no loss of generality since we can just add the cost of one extra wake-up operation to all the energy consumptions.

Before formally proving Theorem 1, we discuss the intuition and the main idea of our reduction. For every gap g_i, $1 \leq i \leq n$, we consider functions of the energy consumed in g_i depending on the load x executed in the gap, see Figure 2.2. Function $f(x) = C + (P(s_{crit})/s_{crit})x$ represents the optimal energy consumption in g_i assuming that the processor transitions to the sleep state in the gap. This consumption does not depend on the gap length, and thus the function is the same for all the gaps. Next consider the energy consumption $h_i(x)$ in g_i assuming that the processor remains in the active state throughout the gap. This consumption depends on the required speed and, using the definition of $P(s)$, is given by an arrangement of three lines. More specifically, $h_i(x) = a_{max}L_i$ for $x \in [0, a_{max}L_i]$ (cf. q_1 in Figure 2.2), $h_i(x) = ((4/9) \cdot (x/L_i) + (5/9)a_{max}) \cdot L_i$ for $x \in (a_{max}L_i, 10a_{max}L_i]$ (shown as q_2), and $h_i(x) = (2(x/L_i) - 15a_{max}) \cdot L_i$ for x in $(10a_{max}L_i, +\infty)$ (not depicted). Function $h_i(x)$ depends on the gap length L_i. Hence, in general, the functions $h_i(x)$, with $1 \leq i \leq n$, are different for the various gaps. For any gap g_i, the optimal energy consumption, with respect to the load executed in it, is given by the lower envelope of f and h_i, represented by the solid line segements in Figure 2.2. Let $\text{LE}_i(x) = \min\{f(x), h_i(x)\}$ denote this lower envelope function.

Assume now that b_i units of J_0's load are executed in gap g_i. Then in g_i an energy of $\mathrm{LE}_i(l_i + b_i)$ is consumed. We can rewrite this as $\mathrm{LE}_i(l_i) + E_i^b(b_i)$, and in this way charge an energy of $\mathrm{LE}_i(l_i)$ to the load l_i and an energy of $E_i^b(b_i)$ to the load b_i, We have $E_i^b(b_i) = LE_i(l_i + b_i) - LE_i(l_i)$. Observe that $\mathrm{LE}_i(l_i)$ is the least possible energy expended for gap g_i and that it is attained for $b_i = 0$ when $E_i^b(b_i) = 0$. Since the $\mathrm{LE}_i(l_i)$ energy units charged to the l_i's depend only on the gaps and l_i's themselves, the goal of any algorithm is to minimize $\sum_{i=1}^n E_i^b(b_i)$ subject to the constraint $\sum_{i=1}^n b_i = B$. In other words, the goal is to distribute the B load units to the gaps g_i, $1 \leq i \leq n$, in a way minimizing the energy charged to them.

The average energy consumption per load unit for the b_i's corresponds to the slope of the line passing through $(l_i, \mathrm{LE}_i(l_i))$ and $(l_i{+}b_i, \mathrm{LE}_i(l_i{+}b_i))$. The key idea of the transformation is that this slope gets minimized when $l_i + b_i = a_{max} L_i$ or, equivalently, when $b_i = a_i$. This minimum possible attainable slope is $1/(2a_{max})$, which is independent of the respective gap g_i. The thick dashed line denoted by q_3 in Figure 2.2 is exactly this line passing through $(l_i, \mathrm{LE}_i(l_i))$ and $(l_i + b_i, \mathrm{LE}_i(l_i + b_i))$, when $b_i = a_i$.

It follows that the total energy charged to the B load units of J_0 is minimized when each b_i is either 0 or a_i. Calculations show that in this case the average energy consumption per load unit is minimized to $1/(2a_{max})$ and hence the total energy charged to the load of J_0, is $B/(2a_{max})$. If there exists at least one gap g_i with $0 < b_i < a_i$ or $b_i > a_i$, then by our construction the slope of the line passing through $(l_i, \mathrm{LE}_i(l_i))$ and $(l_i + b_i, \mathrm{LE}_i(l_i + b_i))$ is greater than $1/(2a_{max})$ which implies that the average energy charged to the load of J_0 is strictly greater than $1/(2a_{max})$, and in turn the total energy consumption is strictly greater than $B/(2a_{max})$.

Formally, our reduction satisfies the following lemma, establishing Theorem 1.

Lemma 1. *There exists a feasible schedule for $\mathcal{I}_S$ that consumes energy of at most $5(n+1)\epsilon a_{max} + nC + \frac{1}{2}\sum_{i=1}^n l_i + \frac{B}{2a_{max}}$ if and only if A of $\mathcal{I}_P$ admits a partition.*

Proof. [of Lemma 1] ($\Leftarrow$) Assume that A in $\mathcal{I}_P$ admits a partition. We show how to construct a feasible schedule for $\mathcal{I}_S$ with an energy consumption of exactly $5(n+1)\epsilon a_{max} + nC + \frac{1}{2}\sum_{i=1}^n l_i + \frac{B}{2a_{max}}$. Let A' be the respective subset in the solution of $\mathcal{I}_P$, and assume that $|A'| = m$. Schedule each job of $\mathcal{J}_2$ at a speed of s_{crit}. This fills the respective execution interval of the job. The total energy consumed by all the jobs of $\mathcal{J}_2$ is $(n+1)\epsilon P(s_{crit}) = 5(n+1)\epsilon a_{max}$. Next, in the gaps g_i such that $a_i \in A'$, execute J_0 and the respective jobs of $\mathcal{J}_1$. We will show that this can be done in a balanced way, so that all the processing volume gets executed at a constant speed of a_{max}. We first observe that any job of $\mathcal{J}_1$ alone has a density less than a_{max} in its execution interval. It therefore remains to show that the total density of the jobs $J_i \in \mathcal{J}_1$, with $a_i \in A'$, and J_0, restricted to the gaps g_i with $a_i \in A'$, is a_{max}. This density is

$$\frac{\sum\limits_{i:a_i\in A'} l_i + B}{\sum\limits_{i:a_i\in A'} L_i} = \frac{a_{max}\sum\limits_{i:a_i\in A'} L_i - B + B}{\sum\limits_{i:a_i\in A'} L_i} = a_{max},$$

as claimed. Finally, we run the jobs $J_i \in \mathcal{J}_1$, with $a_i \notin A'$, at a speed of $s_{crit} = 10a_{max}$ starting directly at their release time. We then transition the processor to the sleep state for the rest of the respective gap. This is feasible because $(L_i a_{max} - a_i)/(10a_{max}) < L_i$. Therefore, the energy expended for J_0, the jobs in $\mathcal{J}_1$ and the wake-up operations is equal to

$$P(a_{max}) \cdot \sum_{i:a_i\in A'} L_i + P(s_{crit}) \cdot \sum_{i:a_i\notin A'} \frac{l_i}{s_{crit}} + (n-m)C =$$
$$a_{max} \sum_{i:a_i\in A'} L_i + \frac{1}{2} \sum_{i:a_i\notin A'} l_i + (n-m)C.$$

It thus suffices to show that

$$mC + \frac{1}{2} \sum_{i:a_i\in A'} l_i + \frac{B}{2a_{max}} = a_{max} \sum_{i:a_i\in A'} L_i,$$

which is equivalent to

$$ma_{max} + \frac{B}{2a_{max}} - \frac{B}{2} = \frac{1}{2}a_{max} \sum_{i:a_i\in A'} L_i.$$

The latter equation holds true because $a_{max} \sum_{i:a_i\in A'} L_i = 2a_{max}m - B + B/a_{max}$.

($\Rightarrow$) Assume now that no solution to $\mathcal{I}_P$ exists. That is, for all subsets $A' \subseteq A$, it holds that $\sum_{a_i\in A'} a_i \neq \sum_{a_i\in A\setminus A'} a_i$. We will show that an optimal schedule for $\mathcal{I}_S$ consumes energy strictly greater than $5(n+1)\epsilon a_{max} + nC + \frac{1}{2}\sum_{i=1}^{n} l_i + \frac{B}{2a_{max}}$. We first argue that there exists an optimal schedule that executes the jobs of $\mathcal{J}_2$ during their whole execution intervals at a speed of s_{crit}. So let $\mathcal{S}$ be any optimal schedule. If no portion of J_0 is processed during the execution intervals of jobs of $\mathcal{J}_2$, there is nothing to show. If a portion of J_0 is executed in such an interval I, then we can modify $\mathcal{S}$ without increasing the total energy consumption: In I an average speed higher than s_{crit} must be used. In the schedule there must exist a gap g_i in which (a) the processor transitions to the sleep state or (b) an average speed less than s_{crit} is used. In the first case we execute a portion of J_0 originally scheduled in I at speed s_{crit} in g_i. This can be done immediately before the processor transitions to the sleep state. The total energy does not increase. In the second case we process a portion of J_0 in g_i by slightly raising the processor speed up to a value of at most s_{crit}. By convexity of the power function, the modified schedule consumes a strictly smaller amount of energy. These schedule modifications can be repeated until the jobs of $\mathcal{J}_2$ are processed exclusively in their execution intervals.

In the following, let $\mathcal{S}$ be an optimal schedule in which the jobs of $\mathcal{J}_2$ are executed at speed s_{crit} in their execution intervals, incurring an energy of $5(n+1)\epsilon a_{max}$. It remains to show that the energy consumed by the wake-up operations, the processing of J_0 and of the jobs in $\mathcal{J}_1$ is strictly greater than $nC + \frac{1}{2}\sum_{i=1}^{n} l_i + \frac{B}{2a_{max}}$.

Assume that $\mathcal{S}$ executes b_i units of J_0's processing volume in gap g_i, $1 \le i \le n$. It holds that $\sum_{i=1}^{n} b_i = B$. For each gap there is a lower bound threshold on the processing volume required so that it is worthwhile not to transition the processor to the sleep state in between. We argue that, for gap g_i, this threshold is $l_i + a_i/a_{max}$: The energy consumed in g_i if jobs are processed at speed s_{crit} and a transition to the sleep state is made equals

$$C + \frac{1}{2}\left(l_i + \frac{a_i}{a_{max}}\right) = \frac{1}{2}2a_{max} + \frac{1}{2}a_{max}L_i - \frac{1}{2}a_i + \frac{1}{2}\frac{a_i}{a_{max}} = a_{max}L_i.$$

If no transition to the sleep state is made, the energy consumption is

$$L_i \cdot P\left(\frac{l_i + a_i/a_{max}}{L_i}\right) = L_i \cdot P\left(a_{max} - \frac{a_i - a_i/a_{max}}{L_i}\right) = a_{max}L_i,$$

which is the same value.

Let $A' \subseteq A$ contain the a_i's such that $b_i \ge a_i/a_{max}$, and let again $|A'| = m$. We assume that the processing volume handled in the g_i's, with $a_i \in A'$, is executed at a uniform speed equal to

$$\frac{\sum\limits_{i:a_i \in A'} (l_i + b_i)}{\sum\limits_{i:a_i \in A'} L_i}.$$

This might not be feasible but, due to the convexity of the power function, the resulting energy consumption is in no case higher than the energy consumption of the original schedule $\mathcal{S}$. Hence in the gaps g_i with $a_i \in A'$ the energy consumption is at least

$$\sum_{i:a_i\in A'} L_i \cdot P\left(\frac{\sum\limits_{i:a_i\in A'}(l_i+b_i)}{\sum\limits_{i:a_i\in A'} L_i}\right)$$
$$= \sum_{i:a_i\in A'} L_i \cdot P\left(a_{max} + \frac{\sum\limits_{i:a_i\in A'} b_i - \sum\limits_{a_i\in A'} a_i}{\sum\limits_{i:a_i\in A'} L_i}\right). \tag{2.1}$$

In the gaps g_i with $a_i \notin A'$, the processor executes jobs at speed s_{crit} and transitions to the sleep state. In these gaps the total energy consumption is

$$(n-m)C + \frac{1}{2}\sum_{i:a_i\notin A'}(l_i+b_i). \tag{2.2}$$

We have to prove that the total energy consumption of (2.1) and (2.2) is strictly greater than $nC + \frac{1}{2}\sum_{i=1}^{n} l_i + \frac{B}{2a_{max}}$. Thus we have to show that

$$\frac{1}{2}\sum_{i=1}^{n} l_i + \frac{B}{2a_{max}} < -mC + \frac{1}{2}\sum_{i:a_i\notin A'} l_i + \frac{1}{2}\sum_{i:a_i\notin A'} b_i$$
$$+ \sum_{i:a_i\in A'} L_i \cdot P\left(a_{max} + \frac{\sum\limits_{i:a_i\in A'} b_i - \sum\limits_{a_i\in A'} a_i}{\sum\limits_{i:a_i\in A'} L_i}\right),$$

which is equivalent to

$$mC + \frac{1}{2}\sum_{i:a_i\in A'} l_i + \frac{B}{2a_{max}}$$
$$< \frac{1}{2}\sum_{i:a_i\notin A'} b_i + \sum_{i:a_i\in A'} L_i \cdot P\left(a_{max} + \frac{\sum\limits_{i:a_i\in A'} b_i - \sum\limits_{a_i\in A'} a_i}{\sum\limits_{i:a_i\in A'} L_i}\right).$$

We consider two distinct cases.
Case (1): Suppose that $\sum_{i:a_i\in A'} b_i \le \sum_{a_i\in A'} a_i$.
Since in this case the argument of P in the above inequality is at most $a_{\max}$, we have to show

$$ma_{max} + \frac{1}{2}\sum_{i:a_i\in A'} l_i + \frac{B}{2a_{max}} < \frac{1}{2}\sum_{i:a_i\notin A'} b_i + a_{max}\sum_{i:a_i\in A'} L_i.$$

Substituting l_i we get

$$m a_{max} - \frac{1}{2}\sum_{a_i \in A'} a_i + \frac{B}{2a_{max}} < \frac{1}{2}\sum_{i:a_i \notin A'} b_i + \frac{1}{2}a_{max}\sum_{i:a_i \in A'} L_i.$$

We then substitute L_i and have

$$-\frac{1}{2}\sum_{a_i \in A'} a_i + \frac{B}{2a_{max}} < \frac{1}{2}\sum_{i:a_i \notin A'} b_i - \frac{1}{2}\frac{a_{max}-1}{a_{max}}\sum_{a_i \in A'} a_i,$$

which is equivalent to

$$\frac{B}{2a_{max}} < \frac{1}{2}\sum_{i:a_i \notin A'} b_i + \frac{1}{2a_{max}}\sum_{a_i \in A'} a_i.$$

If $\sum_{i:a_i \notin A'} b_i = 0$, then $\sum_{i:a_i \in A'} b_i = B$. Since by our assumption $\sum_{a_i \in A'} a_i \neq B$, it must be the case that $B = \sum_{i:a_i \in A'} b_i < \sum_{a_i \in A'} a_i$ and the inequality follows.

If on the other hand $\sum_{i:a_i \notin A'} b_i = X > 0$, we have $\sum_{i:a_i \in A'} b_i = B - X \leq \sum_{a_i \in A'} a_i$, and we wish to show

$$\frac{B}{2a_{max}} < \frac{1}{2}X + \frac{B}{2a_{max}} - \frac{X}{2a_{max}}.$$

The above holds for any $X > 0$ and $a_{max} \geq 2$.

Case (2): Suppose that $\sum_{i:a_i \in A'} b_i > \sum_{a_i \in A'} a_i$.
Let $\sum_{i:a_i \notin A'} b_i = X \geq 0$. It follows that $\sum_{i:a_i \in A'} b_i = B - X > \sum_{a_i \in A'} a_i$. We wish to show that

$$mC + \frac{1}{2}a_{max}\sum_{i:a_i \in A'} L_i - \frac{1}{2}\sum_{a_i \in A'} a_i + \frac{B}{2a_{max}} < \frac{1}{2}X$$
$$+ \sum_{i:a_i \in A'} L_i \cdot P\left(a_{max} + \frac{B - \sum\limits_{a_i \in A'} a_i}{\sum\limits_{i:a_i \in A'} L_i} - \frac{X}{\sum\limits_{i:a_i \in A'} L_i} \right).$$

Since $(4/9)s + (5/9)a_{max} \leq 2s - 15a_{max}$ for any $s \geq 10a_{max}$, and the argument of P in the above inequality is strictly greater than a_{max}, we may use the middle branch of the power function. The inequality then becomes

$$mC + \frac{1}{2}a_{max}\sum_{i:a_i \in A'} L_i - \frac{1}{2}\sum_{a_i \in A'} a_i + \frac{B}{2a_{max}}$$
$$< \frac{1}{2}X + a_{max}\sum_{i:a_i \in A'} L_i + \frac{4}{9}(B - \sum_{a_i \in A'} a_i) - \frac{4}{9}X,$$

which is equivalent to

$$mC - \frac{1}{2}\sum_{a_i \in A'} a_i + \frac{B}{2a_{max}} < \frac{1}{18}X + \frac{1}{2}a_{max}\sum_{i:a_i \in A'} L_i + \frac{4}{9}(B - \sum_{a_i \in A'} a_i).$$

By substituting L_i, we get

$$-\frac{1}{2}\sum_{a_i \in A'} a_i + \frac{B}{2a_{max}} < \frac{1}{18}X - \frac{1}{2}\sum_{a_i \in A'} a_i + \frac{\sum_{a_i \in A'} a_i}{2a_{max}} + \frac{4}{9}(B - \sum_{a_i \in A'} a_i),$$

or equivalently,

$$\frac{B}{2a_{max}} < \frac{1}{18}X + \frac{\sum_{a_i \in A'} a_i}{2a_{max}} + \frac{4}{9}(B - \sum_{a_i \in A'} a_i).$$

It suffices to show that $B/(2a_{max}) < (\sum_{a_i \in A'} a_i)/(2a_{max}) + (4/9)(B - \sum_{a_i \in A'} a_i)$, which holds for $a_{max} \geq 2$. The proof is complete. $\square$

We next present two lower bounds that hold true for all algorithms, independently of their running times. We exploit properties of schedules but do not take into account their construction time. Again, formally a schedule $\mathcal{S}$ for a job set $\mathcal{J}$ is an *s_{crit}-schedule* if any job is processed at a speed of at least s_{crit}.

Theorem 2. *Let A be an algorithm that computes s_{crit}-schedules for any job instance. Then A does not achieve an approximation factor smaller than 2, for general convex power functions.*

Proof. Let ϵ, where $0 < \epsilon < 1$, be an arbitrary constant. We show that A cannot achieve an approximation factor smaller than $2 - \epsilon$. Set $\epsilon' = \epsilon/7$. Fix an arbitrary critical speed $s_{crit} > 0$ and associated power $P(s_{crit}) > 0$. Let $P(0) = \epsilon' P(s_{crit})$. We define a power function $P(s)$ for which s_{crit} is indeed the critical speed. Function $P(s)$ is piecewise linear. In the interval $[0, \epsilon' s_{crit}]$ it is given by the line passing through the points $(0, P(0))$ and $(\epsilon' s_{crit}, (1+\epsilon')P(0))$. In the interval $(\epsilon' s_{crit}, s_{crit})$ it is defined by the line through $(\epsilon' s_{crit}, (1+\epsilon')P(0))$ and $(s_{crit}, P(s_{crit}))$. This line has a slope of $(P(s_{crit}) - (1+\epsilon)P(0))/((1-\epsilon')s_{crit})$. For $s \geq s_{crit}$, $P(s)$ is given by the line $P(s_{crit})s/s_{crit}$. In summary,

$$P(s) = \begin{cases} P(0)(\frac{s}{s_{crit}} + 1), & s \leq \epsilon' s_{crit}, \\ \frac{P(s_{crit}) - (1+\epsilon')P(0)}{1-\epsilon'}(\frac{s}{s_{crit}} - 1) + P(s_{crit}), & \epsilon' s_{crit} < s < s_{crit}, \\ P(s_{crit})\frac{s}{s_{crit}}, & s_{crit} \leq s. \end{cases}$$

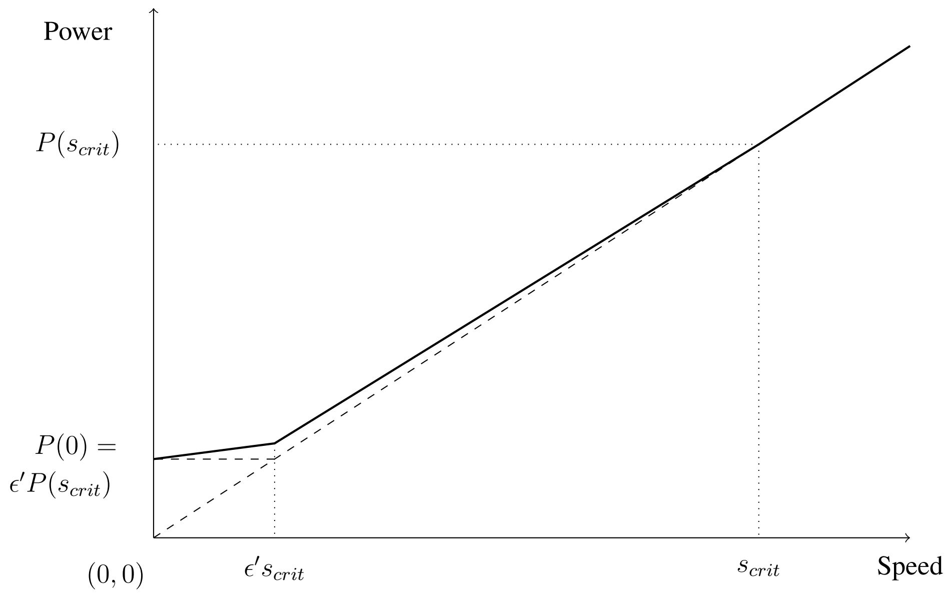


Figure 2.3: The power function $P(s)$.

This power function is increasing and convex because the three slopes form a strictly increasing sequence, by our choice of ϵ'. Furthermore, s_{crit} is the smallest value minimizing $P(s)/s$.

We specify a job sequence. We first define three jobs J_1, J_2 and J_3 with the following characteristics. Let $L > 0$ be an arbitrary constant. Job J_1 has a processing volume of $v_1 = \delta L s_{crit}$, where $\delta = (\epsilon')^2/2$, and can be executed in the interval $I_1 = [0, \delta L)$, i.e. $r_1 = 0$ and $d_1 = \delta L$. The second job J_2 has a processing volume of $v_2 = \epsilon' L s_{crit}$ and can be processed in $I_2 = [\delta L, (1+\delta)L)$ so that $r_2 = \delta L$ and $d_2 = (1+\delta)L$. The third job J_3 is similar to the first one with $v_3 = \delta L s_{crit}$. The job can be executed in $I_3 = [(1+\delta)L, (1+2\delta)L)$. i.e. $r_3 = (1+\delta)L$ and $d_3 = (1+2\delta)L$. The three intervals I_1, I_2 and I_3 are disjoint, and each of the three jobs can be feasibly scheduled using a speed of s_{crit}. Let $C = LP(0)$ be the energy of a wake-up operation.

We analyze the energy consumption of A and an optimal solution, assuming for the moment that the processor is in the active state at time 0. First consider the energy consumption of A. Suppose that A processes some job at a speed higher than s_{crit}. Since $P(s)$ is linear for $s \geq s_{crit}$, we can reduce the speed to s_{crit} without increasing the processing energy needed for the job. The speed reduction only reduces the time while the processor does not execute jobs and thus the idle energy of the schedule. Hence we may analyze A assuming that all the three jobs are processed at speed s_{crit}. Jobs J_1 and J_3 each consume an energy of $\delta L P(s_{crit})$. In I_2 job J_2 is processed for $v_2/s_{crit} = \epsilon' L$ time units. During the remaining time

$L - \epsilon' L = (1-\epsilon')L$ time units the processor is idle. Since $C > (1-\epsilon')LP(0)$, it is not worthwhile to power down and the processor should remain in the active state. Hence A's energy consumption is at least

$$\begin{aligned} 2\delta LP(s_{crit}) + \epsilon' LP(s_{crit}) + (1-\epsilon')LP(0) &> L(\epsilon' P(s_{crit}) + (1-\epsilon')P(0)) \\ &= (2-\epsilon')LP(0). \end{aligned}$$

The last equation holds because $\epsilon' = P(0)/P(s_{crit})$.

In an optimal solution jobs J_1 and J_3 must also be executed at speed s_{crit}. However J_2 can be processed using speed $v_2/L = \epsilon' s_{crit}$ in I_2 so that the energy consumption for the job is $LP(\epsilon' s_{crit}) = (1+\epsilon')LP(0)$. Hence the optimum power consumption is upper bounded by

$$\begin{aligned} 2\delta LP(s_{crit}) + (1+\epsilon')LP(0) &= (\epsilon')^2 LP(s_{crit}) + (1+\epsilon')LP(0) \\ &= (1+2\epsilon')LP(0). \end{aligned}$$

The last equality holds again because $\epsilon' = P(0)/P(s_{crit})$.

Now assume that the processor is in the sleep state initially and a wake-up operation must be performed at time 0. In order to deal with this extra cost of C, we repeat the above job sequence $k = \lceil 1/\epsilon' \rceil$ times. In the i-th repetition, $1 \le i \le k$, there exist three jobs J_{i1}, J_{i2} and J_{i3} with processing volumes $v_{ij} = v_j$. $1 \le j \le 3$. The i-th repetition starts at time $t_i = (i-1)(1+2\delta)L$. For this job sequence the ratio of the energy consumed by A to that of an optimal solution is greater than $\frac{k(2-\epsilon')LP(0)}{C+k(1+2\epsilon')LP(0)} \ge \frac{2-\epsilon'}{1+3\epsilon'} > 2-\epsilon$. The first inequality holds because $C/k \le \epsilon' LP(0)$, and the second one follows because $\epsilon' = \epsilon/7$. $\square$

For the problem instance defined in the above proof, s_{crit}-schedules minimize the processing energy. We obtain:

Corollary 1. *Let A be an algorithm that, for any job instance, computes a schedule minimizing the processing energy. Then A does not achieve an approximation factor smaller than 2, for general convex power functions.*

2.2 A $4/3$-approximation algorithm

We develop a polynomial time $4/3$-approximation algorithm, for general convex power functions. As we will see, the algorithm is an instance of a more general algorithmic framework.

2.2.1 Description of the algorithm

Our general algorithm combines *YDS* and *BCD* while making crucial use of a new, specific speed level s_0 that determines when to switch from *YDS* to *BCD*. For varying s_0, $0 \leq s_0 \leq s_{crit}$, we obtain a family of algorithms *ALG*(s_0). The best choice of s_0 depends on the power function. In order to achieve a $4/3$-approximation for general convex power functions, we choose s_0 such that $P(s_0)/s_0 = \frac{4}{3}P(s_{crit})/s_{crit}$.

We first argue that our speed level s_0, satisfying $P(s_0)/s_0 = \frac{4}{3}P(s_{crit})/s_{crit}$, is well defined. Speed s_{crit} is the smallest value minimizing $P(s)/s$, see [39]. Speed s_{crit} is well defined if $P(s)/s$ does not always decrease, for $s > 0$. If $P(s)/s$ always decreases, then by scheduling each job at infinite speed, or the maximum allowed speed, one obtains trivial optimal schedules. We therefore always assume that there exists a finite speed s_{crit}.

Consider the line $f(s) = P(s_{crit})s/s_{crit}$ with slope $P(s_{crit})/s_{crit}$ passing through $(0,0)$. This line meets the power function $P(s)$ at point $(s_{crit}, P(s_{crit}))$, see Figure 2.4. In fact $f(s)$ is the tangent to $P(s)$ at s_{crit} (assuming that $P(s)$ is differentiable at s_{crit}) since otherwise $P(s_{crit}+\epsilon)/(s_{crit}+\epsilon) < P(s_{crit})/s_{crit}$, for some $\epsilon > 0$, and s_{crit} would not be the true critical speed. Moreover, $P(s)$ is strictly above $f(s)$ in the interval $(0, s_{crit})$, i.e. $P(s) > f(s)$ for all $s \in (0, s_{crit})$, because s_{crit} is the smallest value minimizing $P(s)/s$. Next consider the line $g(s) = \frac{4}{3}f(s) = \frac{4}{3}P(s_{crit})s/s_{crit}$. We have $g(s) > f(s)$, for all $s > 0$, and hence $g(s)$ intersects $P(s)$, for some speed in the range $(0, s_{crit})$. Our speed s_0 is chosen as this value satisfying $g(s_0) = P(s_0)$, and therefore $P(s_0)/s_0 = \frac{4}{3}P(s_{crit})/s_{crit}$. We remark that $g(s)$ intersects $P(s)$ only once in $(0, s_{crit})$ because $P(s)$ is convex and $g(s_{crit}) > f(s_{crit}) = P(s_{crit})$.

In the following we present *ALG*(s_0), for $0 \leq s_0 \leq s_{crit}$. Let $J_1, \ldots, J_n$ be the jobs to be processed. The scheduling horizon is $[r_{\min}, d_{\max})$, where $r_{\min} = \min_{1\leq i\leq n} r_i$ is the earliest release time and $d_{\max} = \max_{1\leq i\leq n} d_i$ is the latest deadline of any job. *ALG*(s_0) operates in two phases.

Description of Phase 1: In Phase 1 the algorithm executes *YDS* and identifies job sets to be processed at speeds higher than s_0 according to this strategy. For completeness we describe *YDS*, which works in rounds. At the beginning of a round R, let $\mathcal{J}$ be the set of unscheduled jobs and H be the available scheduling horizon. Initially, prior to the first round, $\mathcal{J} = \{J_1, \ldots, J_n\}$ and $H = [r_{\min}, d_{\max})$. During the round R *YDS* identifies an interval $I_{\max}$ of maximum density. The *density* $\Delta(I)$ of an interval $I = [t, t')$ is defined as $\Delta(I) = \sum_{J_i \in S(I)} v_i/(t'-t)$, where $S(I) = \{J_i \in \mathcal{J} \mid [r_i, d_i) \subseteq I\}$ is the set of jobs to be processed in I. Given a maximum density interval $I_{\max} = [t, t')$, *YDS* schedules the jobs of $S(I_{\max})$ at speed $\Delta(I_{\max})$ in that interval according to the *Earliest-Deadline-First*

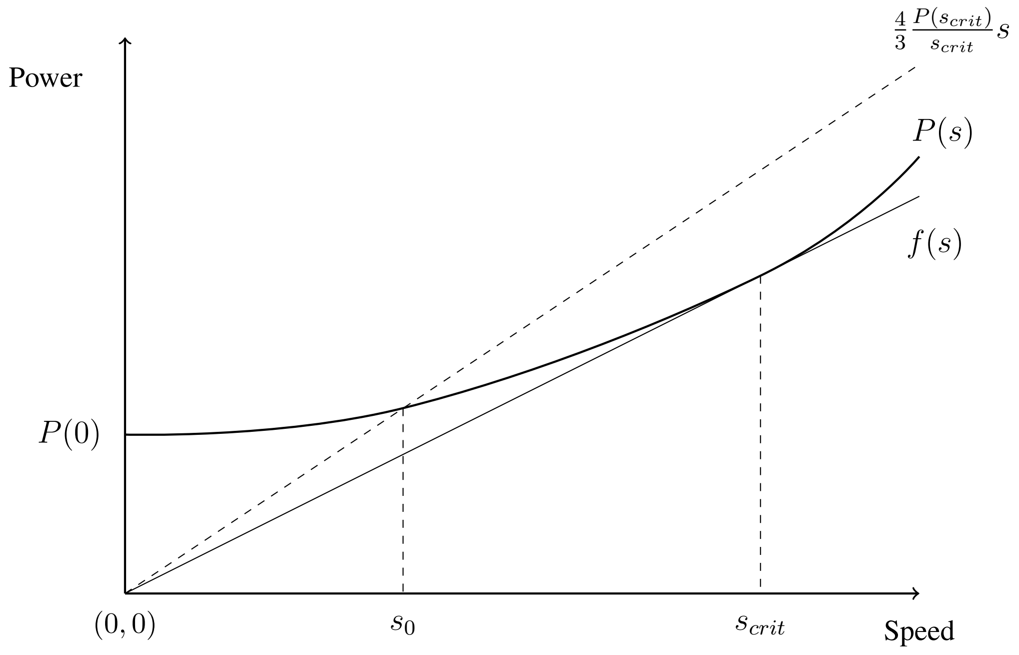


Figure 2.4: The functions $P(s)$, $f(s)$ and $g(s)$.

(EDF) discipline. Then $S(I_{\max})$ is deleted from $\mathcal{J}$, and $I_{\max}$ is removed from H. More specifically, for any unscheduled job $J_i \in \mathcal{J}$ with either $r_i \in I_{\max}$ or $d_i \in I_{\max}$ we set the new release time to $r_i' = t'$ or the new deadline to $d_i' = t$, respectively. Finally, considering the jobs of $\mathcal{J}$, all release times and deadlines of value at least t' are reduced by $t' - t$.

Algorithm *ALG*(s_0) executes scheduling rounds of *YDS* while $\mathcal{J} \neq \emptyset$, and $\Delta(I_{\max}) > s_0$, i.e. jobs are scheduled at speeds higher than s_0. At the end of Phase 1, let $\mathcal{J}_{YDS} \subseteq \{J_1, \dots, J_n\}$ be the set of jobs scheduled according to *YDS*. Considering the original time horizon $[r_{\min}, d_{\max})$, let $I_1, \dots, I_l$ be the sequence of disjoint, non-overlapping intervals in which the jobs of $\mathcal{J}_{YDS}$ are scheduled. These intervals are the portions of $[r_{\min}, d_{\max})$ used by the *YDS* schedule for $\mathcal{J}_{YDS}$. Figure 2.5 depicts an example consisting of five maximum density intervals $I^j_{\max}$, $j = 1, \dots, 5$, forming an interval sequence I_1, I_2. The height of an interval $I^j_{\max}$ corresponds to the density $\Delta(I^j_{\max})$. Given $I_1, \dots, I_l$, let $I_j = [t_j, t_j')$, where $1 \leq j \leq l$. We have $t_j' \leq t_{j+1}$, for $j = 1, \dots, l-1$. We remark that every job of $\mathcal{J}_{YDS}$ is completely scheduled in exactly one interval I_j.

Description of Phase 2: In Phase 2 *ALG*(s_0) constructs a schedule for the set $\mathcal{J}_0 = \{J_1, \dots, J_n\} \setminus \mathcal{J}_{YDS}$ of unscheduled jobs, integrating the partial schedule of Phase 1. The schedule for $\mathcal{J}_0$ uses a uniform speed of s_0 and is computed by properly invoking *BCD*. Algorithm *BCD* takes as input a set of jobs, each specified

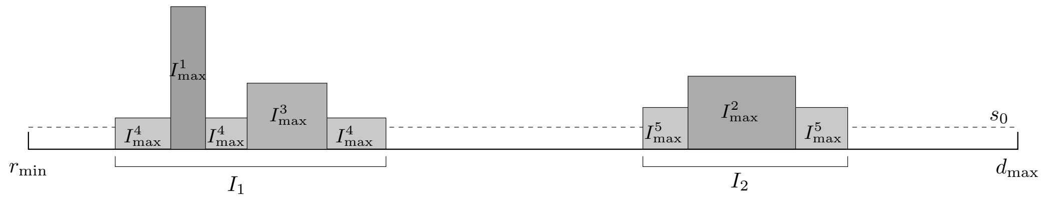


Figure 2.5: Five intervals $I^j_{\max}$, $j = 1, \ldots, 5$, that form I_1 and I_2.

by a release time, a deadline and a processing time. The given processor has an active state and a sleep state. In the active state it consumes 1 energy unit per time unit, even if no job is currently executed. A wake-up operation requires L energy units. *BCD* computes an optimal schedule for the given job set, minimizing energy consumption.

We construct a job set $\mathcal{J}_{BCD}$ to which *BCD* is applied. Initially, we set $\mathcal{J}_{BCD} := \emptyset$. For each $J_i \in \mathcal{J}_0$ we introduce a job J'_i of processing time $v'_i = v_i/s_0$ because in a speed-s_0 schedule J_i has to be processed for v_i/s_0 time units. The execution interval of J'_i is the same as that of J_i, i.e. $r'_i = r_i$ and $d'_i = d_i$. We add J'_i to $\mathcal{J}_{BCD}$. In order to ensure that the jobs J'_i are not processed in the intervals $I_1, \ldots, I_l$, we introduce a job $J(I_j)$ for each such interval $I_j = [t_j, t'_j)$, $1 \leq j \leq l$. Job $J(I_j)$ has a processing time of $t'_j - t_j$, which is the length of I_j, a release time of t_j and a deadline of t'_j. Note that by construction, each job $J(I_j)$, $1 \leq j \leq l$, has to be executed throughout I_j. These jobs $J(I_j)$, $1 \leq j \leq l$, are also added to $\mathcal{J}_{BCD}$.

Using algorithm *BCD*, we compute an optimal schedule for $\mathcal{J}_{BCD}$, assuming that a wake-up operation of the processor incurs $L = C/P(0)$ energy units. Loosely speaking, we normalize energy by $P(0)$ so that, whenever the processor is active and even executes jobs, 1 energy unit per time unit is consumed. Let $\mathcal{S}_{BCD}$ be the schedule obtained. In a final step we modify $\mathcal{S}_{BCD}$: Whenever a job of $\mathcal{J}_{BCD}$ is processed, the speed is set to s_0. Whenever the processor is active but idle, the speed is $s = 0$. The wake-up operations are as specified in $\mathcal{S}_{BCD}$ but incur a cost of C. In the intervals $I_1, \ldots, I_l$ we replace the jobs $J(I_j)$, $1 \leq l \leq l$, by *YDS* schedules for the jobs of $\mathcal{J}_{YDS}$. This schedule is output by our algorithm. A pseudo-code description is given in Figure 2.6. Obviously, *ALG*(s_0) has polynomial running time.

Theorem 3. *Setting s_0 such that $P(s_0)/s_0 = \frac{4}{3}P(s_{crit})/s_{crit}$, ALG$(s_0)$ achieves an approximation factor of $4/3$, for general convex power functions.*

One remark is in order here. Algorithm *BCD* assumes that time is slotted and all processing times, release times and deadlines are integers. This is no loss of generality if problem instances, in a computer, are encoded using rational numbers. If one insists on working with real numbers, in Phase 2 of *ALG*(s_0) *BCD*

Algorithm ALG(s_0):

Phase 1:
Let $\mathcal{J} = \{J_1, \ldots, J_n\}$. While $\mathcal{J} \neq \emptyset$ and the maximum density interval $I_{\max}$ satisfies $\Delta(I_{\max}) > s_0$, execute *YDS*. At the end of the phase let $\mathcal{J}_{YDS}$ be the set of jobs scheduled according to *YDS* and $I_1, \ldots, I_l$ be the sequence of intervals used. Let $\mathcal{J}_0 = \{J_1, \ldots, J_n\} \setminus \mathcal{J}_{YDS}$.

Phase 2:
Let $\mathcal{J}_{BCD} = \emptyset$. For any $J_i \in \mathcal{J}_0$, add a job J'_i with processing time $v'_i = v_i/s_0$. release time $r'_i = r_i$ and deadline $d'_i = d_i$ to $\mathcal{J}_{BCD}$. For each I_j, $1 \leq j \leq l$, add a job $J(I_j)$ with processing time $t'_j - t_j$, release time t_j and deadline t'_j to $\mathcal{J}_{BCD}$. Compute an optimal schedule $\mathcal{S}_{BCD}$ for $\mathcal{J}_{BCD}$ using *BCD* and assuming that a wake-up operation incurs $C/P(0)$ energy units. In this schedule, set the speed to s_0 whenever a job of $\mathcal{J}_{BCD}$ is processed. In the intervals $I_1, \ldots, I_l$ replace $J(I_j)$, for $1 \leq j \leq l$, by *YDS* schedules for $\mathcal{J}_{YDS}$.

Figure 2.6: The algorithm *ALG*(s_0), where $0 \leq s_0 \leq s_{crit}$.

can compute optimal solutions to an arbitrary precision. In this case our algorithm achieves an approximation factor of $4/3 + \epsilon$, for any $\epsilon > 0$. In the following we assume that the input is encoded using rational numbers.

2.2.2 Analysis of the algorithm

We analyze *ALG*(s_0) and prove Theorem 3. Let $\mathcal{J} = \{J_1, \ldots, J_n\}$ denote the set of all jobs to be scheduled. Furthermore, let $\mathcal{S}_A$ be the schedule constructed by *ALG*(s_0). Let $\mathcal{S}$ be any feasible schedule for $\mathcal{J}$ and $\mathcal{J}' \subseteq \mathcal{J}$ be any subset of the jobs. We say that *$\mathcal{S}$ schedules $\mathcal{J}'$ according to YDS* if, considering the time intervals in which the jobs of $\mathcal{J}'$ are processed, the corresponding partial schedule is identical to the schedule constructed by *YDS* for $\mathcal{J}'$, assuming that *YDS* starts from an initially empty schedule. In Phase 1 *ALG*(s_0) schedules job set $\mathcal{J}_{YDS} \subseteq \mathcal{J}$ according to *YDS*. Let $\mathcal{J}'_{YDS} \subseteq \mathcal{J}_{YDS}$ be the set of jobs that are processed at speeds higher than s_{crit}. Irani et al. [39] showed that there exists an optimal schedule for the entire job set $\mathcal{J}$ that schedules $\mathcal{J}'_{YDS}$ according to *YDS*. In the following let $\mathcal{S}_{OPT}$ be such an optimal schedule.

For the further analysis we transform $\mathcal{S}_{OPT}$ into a schedule $\mathcal{S}_0$ that will allow us to compare $\mathcal{S}_A$ to $\mathcal{S}_{OPT}$. Schedule $\mathcal{S}_0$ schedules $\mathcal{J}_{YDS}$ according to *YDS* and all other jobs at speed s_0. The schedule has the specific feature that its idle energy does not increase too much, compared to that of $\mathcal{S}_{OPT}$. We will prove

$$E(\mathcal{S}_A) \leq E(\mathcal{S}_0) \leq \tfrac{4}{3} E(\mathcal{S}_{OPT}), \tag{2.3}$$

which establishes Theorem 3. The first part of (2.3) holds for any s_0 with $0 \leq s_0 \leq s_{crit}$. The second part holds for our choice of s_0 as specified in Theorem 3.

Transforming the optimal schedule: We describe the algorithm *Trans* that performs the transformation, for any $0 \leq s_0 \leq s_{crit}$. The transformation consists of four steps. In overview, in the first step, the processor speeds of jobs of $\mathcal{J}_0$ that are originally executed at speeds lower than s_0 are raised to s_0. Then, in the second step, $\mathcal{J}_{YDS}$ is scheduled in $I_1, \ldots, I_l$ using *YDS*. In the third and fourth steps, all jobs of $\mathcal{J}_0$ are scheduled at speed exactly s_0. In the original schedule some jobs of $\mathcal{J}_0$ might be processed at speeds higher than s_0. In order to obain a feasible schedule, in which all jobs of $\mathcal{J}_0$ are processed using speed s_0, we may have to resort to times where the processor is idle or in the sleep state in $\mathcal{S}_{OPT}$.

Step 1: Given $\mathcal{S}_{OPT}$, *Trans* first raises processor speeds to s_0. For any job $J_i \in \mathcal{J}_0 = \mathcal{J} \setminus \mathcal{J}_{YDS}$ that is processed at a speed smaller than s_0, the speed is raised to s_0. The processing of J_i can be done at any time in the intervals reserved for J_i in $\mathcal{S}_{OPT}$. This speed increase generates processor idle times. At those times the processor remains in the active state. The state transitions of the schedule are not affected. Let $\mathcal{S}_{0,1}$ be the schedule obtained after this step. We will use this schedule later when analyzing idle energy.

Step 2: Next *Trans* schedules $\mathcal{J}_{YDS}$ according to *YDS* in the intervals $I_1, \ldots, I_l$. At the beginning of an interval $I_j = [t_j, t'_j)$ a wake-up operation has to be performed if the processor is originally in the sleep state at time t_j. Similarly, a power-down operation is performed at the end of the interval if the processor is in the sleep state at time t'_j.

The major part of the transformation consists in scheduling the jobs of $\mathcal{J}_0$ at the remaining times. In the scheduling horizon $[r_{\min}, d_{\max})$, let $I'_1, \ldots, I'_{l'}$ be the time intervals not covered by $I_1, \ldots, I_l$, i.e. the sequences $I_1, \ldots, I_l$ and $I'_1, \ldots, I'_{l'}$ form a partition of $[r_{\min}, d_{\max})$. Within $I'_1, \ldots, I'_{l'}$, let $T_1, \ldots, T_m$ be the sequence of intervals in which the processor resides in the active state in $\mathcal{S}_{OPT}$ and hence in $\mathcal{S}_{0,1}$. The processor may or may not execute jobs at those times. Each T_k is a subinterval of some I'_j, for $1 \leq k \leq m$ and $1 \leq j \leq l'$. An interval I'_j may contain several T_k if the processor transitions to the sleep state in between. An interval I'_j does not contain any T_k if the processor resides in the sleep state throughout I'_j. In order to schedule $\mathcal{J}_0$ in $I'_1, \ldots, I'_{l'}$, *Trans* performs two passes over the schedule. The speed used for any job of $\mathcal{J}_0$ is equal to s_0. In the first pass (Step 3) *Trans* assigns as much work as possible to the intervals $T_1, \ldots, T_m$. In a second pass (Step 4) *Trans* constructs a feasible schedule for $\mathcal{J}_0$, resorting to times at which the processor is in the sleep state. Jobs are always scheduled according to the

Earliest Deadline First (EDF) discipline.

Step 3: In the first pass *Trans* sweeps over the intervals $T_1, \ldots, T_m$ and constructs an *EDF schedule* for $\mathcal{J}_0$, i.e. at any time it schedules a job having the earliest deadline among the available unfinished jobs in $\mathcal{J}_0$. A job J_i is *available* at any time t if $t \in [r_i, d_i)$. Let $\mathcal{S}_{0,3}$ denote the schedule obtained after this step.

The schedule obtained after the first pass might not be feasible in that some jobs are not processed completely. The intervals $T_1, \ldots, T_m$ might be too short to execute all jobs of $\mathcal{J}_0$ at speed s_0, considering in particular the times when the jobs are available for processing. In the second pass *Trans* schedules the remaining work. After the first pass, let p_i be the total time for which $J_i \in \mathcal{J}_0$ is processed in the current schedule. Let $\delta_i = v_i/s_0 - p_i$ be the remaining time for which J_i has to be executed.

Step 4: In the second pass *Trans* considers the jobs of $\mathcal{J}_0$ in non-decreasing order of deadlines; ties may be broken arbitrarily. For each $J_i \in \mathcal{J}_0$ with $\delta_i > 0$, the following steps are executed. In the current schedule let τ_i, where $\tau_i < d_i$, be the last point of time such that $[\tau_i, d_i)$ contains exactly δ_i time units at which the processor does not execute jobs. In a correctness proof given in Lemma 2 we will show that τ_i is well defined. We distinguish two cases.

Case (a): If the processor resides in the sleep state throughout $[\tau_i, d_i)$, then we can easily modify the schedule. Let τ'_i, where $\tau'_i < \tau_i$, be the most recent time when the processor transitions to the sleep state. In Lemma 2 we will also prove that $\tau'_i \geq r_i$. *Trans* modifies the schedule by processing J_i in the interval $[\tau'_i, \tau'_i + \delta_i)$; then the processor is transitioned to the sleep state.

Case (b): If the processor resides in the active state at any time in $[\tau_i, d_i)$, then *Trans* constructs an *EDF* schedule for the remaining work of J_i and the work of $\mathcal{J}_0$ currently processed in $[\tau_i, d_i)$. Formally, *Trans* constructs a small problem instance $\mathcal{W}$ representing the work of $\mathcal{J}_0$ to be done in $[\tau_i, d_i)$. For each job $J_k \in \mathcal{J}_0$ that is currently processed for p'_k time units in $[\tau_i, d_i)$, *Trans* adds a job J'_k of processing volume $v'_k = p'_k s_0$ to $\mathcal{W}$. The job's release time and deadline are the same as those of J_k, i.e. $r'_k = r_k$ and $d'_k = d_k$. If $\mathcal{W}$ already contains a job J'_i associated with J_i, we increase v'_i by $\delta_i s_0$. Otherwise a new job J'_i of processing volume $v'_i = \delta_i s_0$, release time $r'_i = r_i$ and deadline $d'_i = d_i$ is added to $\mathcal{W}$. *Trans* then generates an *EDF* schedule for $\mathcal{W}$ in $[\tau_i, d_i)$, ignoring the intervals $I_1, \ldots, I_l$ that may be contained in $[\tau_i, d_i)$. Again, in Lemma 2 we will show that this is always possible. In the modified schedule the processor executes jobs throughout $[\tau_i, d_i)$ and any wake-up operations performed within the interval are canceled. However, a wake-up operation must be performed at time τ_i if the processor is in the sleep state immediately before time τ_i. Similarly, the processor powers down at the end of $[\tau_i, d_i)$ if the processor is in the sleep state at time d_i.

A summary of *Trans* is given in Figure 2.7.

Algorithm Trans:

1. Given $\mathcal{S}_{OPT}$, for any job of $\mathcal{J}_0$ processed at a speed smaller than s_0, raise the speed to s_0. Let $\mathcal{S}_{0,1}$ be the resulting schedule.
2. In $I_1, \ldots, I_l$ schedule $\mathcal{J}_{YDS}$ according to *YDS*. Let $T_1, \ldots, T_m$ be the sequence of intervals within $I'_1, \ldots, I'_{l'}$, in which the processor resides in the active state in $\mathcal{S}_{OPT}$.
3. In $T_1, \ldots, T_m$ construct an *EDF* schedule for $\mathcal{J}_0$ using speed s_0. For any $J_i \in \mathcal{J}_0$, let p_i be the time for which J_i is processed and $\delta_i = v_i/s_0 - p_i$.
4. Consider jobs of $\mathcal{J}_0$ in non-decreasing order of deadlines. For each J_i with $\delta_i > 0$, execute the following steps. In the current schedule $\mathcal{S}$, let $[\tau_i, d_i)$ be the shortest interval containing exactly δ_i time units at which the processor does not execute jobs.

   If the processor is in the sleep state throughout $[\tau_i, d_i)$, schedule J_i in $[\tau'_i,\ \tau'_i + \delta_i)$, where τ'_i is the most recent time before τ_i at which the processor powers down.

   If the processor is in the active state at some time in $[\tau_i, d_i)$, then for any $J_k \in \mathcal{J}_0$ that is processed for $p'_k > 0$ time units in $[\tau_i, d_i)$, add an entry (J'_k, r_k, d_k, v'_k) with $v'_k = p'_k s_0$ to $\mathcal{W}$. If $\mathcal{W}$ contains an entry for J_i, increase v'_i by $\delta_i s_0$; otherwise add an entry (J'_i, r_i, d_i, v'_i) with $v'_i = \delta_i s_0$. Construct an *EDF* schedule for $\mathcal{W}$ in $[\tau_i, d_i)$, ignoring the intervals $I_1, \ldots, I_l$ where $\mathcal{J}_{YDS}$ is scheduled.

Figure 2.7: The algorithm *Trans*.

The following lemma shows correctness of *Trans*.

Lemma 2. *Trans constructs a feasible schedule $\mathcal{S}_0$ in which all jobs of $\mathcal{J}$ are completely scheduled.*

Proof. All jobs of $\mathcal{J}_{YDS}$ are feasibly and completely scheduled in $I_1, \ldots, I_l$. Therefore, it suffices to show that *Trans* constructs a feasible and complete schedule for $\mathcal{J}_0$. For ease of exposition we now black out $I_1, \ldots, I_l$ in the scheduling horizon since these intervals are not used in Steps 3 and 4 of *Trans*. Similarly, we ignore $\mathcal{J}_{YDS}$. In the condensed time horizon, consisting of $I'_1, \ldots, I'_{l'}$, let r_i and d_i denote the release time and deadline of any $J_i \in \mathcal{J}_0$. By the definition of Phase 1 of *ALG*(s_0), in the condensed time horizon any interval $[t, t')$ has a density of at most s_0.

We first show that the value τ_i can always be feasibly chosen as specified in Step 4 of *Trans*. We number the jobs of $\mathcal{J}_0$ in non-decreasing order of deadlines; ties are broken arbitrarily. In this sequence let J_i be the i-th job, $1 \leq i \leq |\mathcal{J}_0|$. In the following, a schedule $\mathcal{S}$ is referred to as an *EDF schedule for* $\mathcal{J}_0$, if at any time when the processor executes a job, it processes one having the earliest deadline among the available unfinished jobs of $\mathcal{J}_0$. Schedule $\mathcal{S}$ might not process each job $J_i \in \mathcal{J}_0$ completely. We will prove that the following statement holds, for any $i = 1, \ldots, |\mathcal{J}_0|$.

(S) Let $\mathcal{S}$ be an *EDF* schedule for $\mathcal{J}_0$ in which the first $i-1$ jobs of $\mathcal{J}_0$ are processed completely. Then the execution of Step 4 of *Trans* for J_i yields an *EDF* schedule for $\mathcal{J}_0$ in which the first i jobs of $\mathcal{J}_0$ are processed completely.

Lemma 2 then follows because the schedule constructed in Step 3 of *Trans* is an *EDF* schedule for $\mathcal{J}_0$.

Let $\mathcal{S}$ be an *EDF* schedule for $\mathcal{J}_0$ in which the first $i-1$ jobs are processed completely. Consider job J_i with its deadline d_i. If J_i is processed for $p_i = v_i/s_0$ time units, we are done. So suppose $p_i < v_i/s_0$, and let $\delta_i = v_i/s_0 - p_i$ be the additional time for which J_i has to be executed. We first prove the following fact: Let $I = [\tau, d_i)$ be any time interval in $\mathcal{S}$ containing strictly less than δ_i time units at which the processor does not execute jobs. Then in this interval I the processor does not execute any jobs having a deadline after d_i.

We prove the fact by contradiction. Suppose that in I the processor executes a job whose deadline is after d_i. The processing of such a job cannot end at time d_i because $\mathcal{S}$ is an *EDF* schedule and J_i is not completely processed at time d_i. Let τ', with $\tau < \tau' < d_i$, be the earliest time such that in $[\tau', d_i)$ the processor does not execute jobs having a deadline after d_i. All the jobs J_k, with $k \neq i$, executed in $[\tau', d_i)$ have a release time of at least τ' because immediately before time τ' a job with a deadline after d_i is processed. By the same argument, J_i has a release time of at least τ'. The total processing volume of J_i and the jobs J_k, with $k \neq i$, executed in $[\tau', d_i)$ is at least $(p + \delta_i)s_0$, where p is the total time for which jobs are executed in $[\tau', d_i)$. However, the latter interval has a length strictly smaller than $p + \delta_i$ because I contains less than δ_i time units at which the processor does not execute any jobs. Hence the density of $[\tau', d_i)$ is strictly higher than s_0, which contradicts the fact that in the scheduling horizon all intervals have a density of at most s_0.

Using the above fact we can easily show that the value τ_i in Step 4 of *Trans* is well defined. Let r_0 be the earliest release time among the jobs of $\mathcal{J}_0$. If in the schedule $\mathcal{S}$ interval $[r_0, d_i)$ contained less than δ_i time units at which the processor does not execute any jobs, then by the above fact only jobs with a deadline of at most d_i can be processed in $[r_0, d_i)$. The total processing volume of J_i and the jobs J_k, with $k \neq i$, executed in $[r_0, d_i)$ is at least $(p + \delta_i)s_0$, where p is the total time for which jobs are executed in $[r_0, d_i)$. We obtain again a contradiction to the fact that the density of $[r_0, d_i)$ is at most s_0. Hence $[r_0, d_i)$ contains at least δ_i time units at which the processor does not execute jobs and a value τ_i, with $\tau_i \geq r_0$, as specified in Step 4 of *Trans* can be feasibly chosen.

We next analyze the schedule modifications of Step 4. First suppose that the processor is in the sleep state throughout $[\tau_i, d_i)$, considering the full time horizon given by $I_1, \dots, I_l$ and $I'_1, \dots, I'_{l'}$. Then let τ'_i, with $\tau'_i < \tau_i$ be the most recent time when the processor transitions to the sleep state. The processor is in the active state immediately before τ'_i. Time τ'_i satisfies $\tau'_i \geq r_i$ because in the original schedule $\mathcal{S}_{OPT}$ job J_i was completely scheduled and the processor must be in the active state at some time while J_i is available for processing. We note that τ'_i might coincide with the end of an interval I_j, $1 \leq j \leq l$. Job J_i is now scheduled for the missing δ_i time units in $[\tau'_i, \tau'_i + \delta_i)$. The modified schedule is an *EDF* schedule for $\mathcal{J}_0$ because no further job is processed in $[\tau'_i + \delta_i, d_i)$.

Next suppose that the processor is in the active state at any time in $[\tau_i, d_i)$, taking again into account the full time horizon. We show that when Step 4 of *Trans* is executed for J_i, all the work of $\mathcal{W}$ is completely scheduled in $[\tau_i, d_i)$. Recall that $\mathcal{S}$ is the schedule prior to the modification. By the choice of τ_i, the processor does not execute any jobs at time τ_i in $\mathcal{S}$. Choose an $\epsilon > 0$ such that the processor does not execute jobs in $[\tau_i, \tau_i + \epsilon)$. Interval $[\tau_i + \epsilon, d_i)$ contains less than δ_i time units at which the processor does not execute jobs. By the above fact, in $[\tau_i + \epsilon, d_i)$ and hence in $[\tau_i, d_i)$, only jobs with a deadline of at most d_i are executed.

Suppose that the execution of Step 4 yielded a schedule $\mathcal{S}'$ in which $[\tau_i, d_i)$ does not allocate all the work of $\mathcal{W}$. Then there must exist a time in that interval at which no job is executed. Let τ, with $\tau > \tau_i$, be the earliest time such that the processor executes jobs throughout $[\tau, d_i)$ in $\mathcal{S}'$. We argue that $\tau < d_i$, i.e. some job is scheduled until time d_i: *Trans* schedules the work $\mathcal{W}$ in $[\tau_i, d_i)$ according to *EDF*. Hence at any time when jobs with a deadline smaller than d_i can be processed, they have preference over the jobs with a deadline equal to d_i. Thus if some work of $\mathcal{W}$ is not allocated to $[\tau_i, d_i)$, the work corresponds to jobs having a deadline of d_i. These jobs can be feasibly scheduled immediately before time d_i. Hence $\tau < d_i$. Consider the jobs J'_k of $\mathcal{W}$ that are executed in $[\tau, d_i)$ in $\mathcal{S}'$ or are not completely allocated to $[\tau_i, d_i)$. All these jobs have a release time of at least τ because the processor does not execute any job immediately before time τ. Moreover, these jobs have a deadline of at most d_i. It follows that the total processing volume of these jobs J'_k and hence of the original jobs J_k is greater than $(d_i - \tau)s_0$, contradicting the fact that $[\tau, d_i)$ has a density of at most s_0.

It remains to show that $\mathcal{S}'$ is an *EDF* schedule for $\mathcal{J}_0$. In the interval $[r_0, \tau_i)$, schedule $\mathcal{S}$ and hence $\mathcal{S}'$ are *EDF* schedules for $\mathcal{J}_0$. In $[\tau_i, d_i)$ both schedules execute jobs having a deadline of at most d_i. In $[d_i, d_0)$, where d_0 denotes the maximum deadline of any job of $\mathcal{J}_0$, jobs with a deadline greater than d_i are scheduled. Schedule $\mathcal{S}'$ represents an *EDF* schedule for $\mathcal{J}_0$ because (a) *Trans* schedules $\mathcal{W}$ according to *EDF* and (b) in $[d_i, d_0)$ schedule $\mathcal{S}$ and hence $\mathcal{S}'$ process the respective workload according to *EDF*. □

Analyzing energy: We first analyze idle energy. In Step 1 of *Trans*, when speeds are raised to s_0, the idle energy increases. We will analyze this increase later. Lemma 3 below implies that Steps 2–4 of *Trans* do not cause a further increase. Recall that $\mathcal{S}_{0,1}$ denotes the schedule obtained after Step 1 of *Trans*. Lemmas 3 and 4 below hold for any speed s_0, with $0 \le s_0 \le s_{crit}$.

Lemma 3. *There holds* $E_i(\mathcal{S}_0) \le E_i(\mathcal{S}_{0,1})$.

Proof. Consider the schedule obtained after Step 3 of *Trans*, which we denote by $\mathcal{S}_{0,3}$. We will show $E_i(\mathcal{S}_{0,3}) \le E_i(\mathcal{S}_{0,1})$. The lemma then follows because the scheduling operations of Step 4 do not increase the idle energy: Consider a job J_i for which Step 4 is executed. If the processor is in the sleep state throughout $[\tau_i, d_i)$, then *Trans* determines the most recent time τ_i' at which the processor transitions to the sleep state. Job J_i is scheduled in the interval $[\tau_i', \tau_i' + \delta_i)$; then the processor powers down. Neither the number of wake-up operations nor the total time for which the processor is in the active state but does not process jobs increase. If the processor is in the active state at some time in $[\tau_i, d_i)$, then the *EDF* schedule generated for this interval does not increase the idle energy, either. The processor might have to transition to the active state at time τ_i. However, this cancels the subsequent wake-up operation needed to transition the processor to the active state in $[\tau_i, d_i)$. Throughout $[\tau_i, d_i)$ the processor executes jobs and no idle energy is incurred.

In order to prove $E_i(\mathcal{S}_{0,3}) \le E_i(\mathcal{S}_{0,1})$ we transform $\mathcal{S}_{0,1}$ into $\mathcal{S}_{0,3}$ without increasing the idle energy. Considering the scheduling horizon $[r_{\min}, d_{\max})$, we sweep over the schedule $\mathcal{S}_{0,1}$ from left to right. At any time t, $r_{\min} \le t \le d_{\max}$, we maintain a schedule $\mathcal{S}_t$. To the left of t, i.e. in $[r_{\min}, t)$, $\mathcal{S}_t$ is identical to $\mathcal{S}_{0,3}$. To the right of t, in $[t, d_{\max})$, the schedule still has to be transformed. During the transformation we maintain a buffer B that contains, for each job $J_i \in \mathcal{J}_0$, an entry (J_i, w_i) representing the amount of work that is not finished for J_i in $\mathcal{S}_t$. We maintain the invariant that, for any $J_i \in \mathcal{J}_0$, the processing volume finished for J_i in $\mathcal{S}_t$ plus w_i is equal to v_i. Jobs of $\mathcal{J}_{YDS}$ are always completely processed in the current schedule and hence need not be represented in the buffer. Initially, at time $t = r_{\min}$, since $\mathcal{S}_{0,1}$ completely schedules all jobs, B contains an entry $(J_i, 0)$, for all $J_i \in \mathcal{J}_0$. While sweeping over $\mathcal{S}_{0,1}$, we consider the full intervals $I_1, \ldots, I_l$. In the sequence $I_1', \ldots, I_l'$ we only consider the intervals $T_1, \ldots, T_m$ in which the processor is active; times at which the processor is in the sleep state can be ignored. Time t is always equal to the beginning of some interval I_j, $1 \le j \le l$, or equal to some time in an interval T_j, $1 \le j \le m$. Initially, t is set to the beginning of I_1 or T_1, depending on which of the two intervals occurs earlier in the scheduling horizon. The initial $\mathcal{S}_t$ is $\mathcal{S}_{0,1}$.

Suppose that we have already constructed a schedule $\mathcal{S}_t$ up to time t, $r_{\min} \le t \le d_{\max}$. If t is equal to the starting point of an interval $I_j = [t_j, t'_j)$, then we modify the schedule as follows. For any job $J_i \in \mathcal{J}_0$ that is processed for δ time units using speed s in I_j, we increase w_i by δs in the buffer B. Then we schedule the jobs of $\mathcal{J}_{YDS}$ to be processed in I_j according to *YDS*. Recall that every job in $\mathcal{J}_{YDS}$ is completely scheduled in exactly one interval I_j. Next we determine the smallest time t', where $t' \ge t'_j$, such that t' is the beginning of an interval I_k, where $1 \le k \le l$, or of an interval T_k, where $1 \le k \le m$. Time t is set to t' and the modified schedule is the new $\mathcal{S}_t$. If no such time t' exists, the transformation is complete.

Next assume that t is a time in some interval T_j, $1 \le j \le m$. Determine the largest δ, $\delta > 0$, such that $[t, t+\delta) \subseteq T_j$ and the following two properties hold: (1) In $\mathcal{S}_{0,3}$ throughout $[t, t+\delta)$ the processor executes the same job J_i or does not execute any job at all. (2) In $\mathcal{S}_t$ throughout $[t, t+\delta)$ the processor executes the same job J_k or does not execute any job at all. Hence in $[t, t+\delta)$ the two schedules do not swap jobs. As we consider an interval T_j, the jobs J_i and J_k possibly executed in $[t, t+\delta)$ belong to $\mathcal{J}_0$: by definition any jobs of $\mathcal{J}_{YDS}$ can only be contained in exactly one of the intervals I_k, $1 \le k \le l$. We have to consider various cases.

Suppose that in $\mathcal{S}_{0,3}$ a job J_i is executed while in $\mathcal{S}_t$ no job is executed, considering the interval $[t, t+\delta)$. Determine the largest ϵ, where $0 \le \epsilon \le \delta$, such that $\epsilon s_0 \le w_i$. In the buffer B we reduce J_i's residual processing volume by ϵs_0. Moreover, in $\mathcal{S}_t$ determine the next $\epsilon' \ge 0$ time units in which a processing volume of $(\delta - \epsilon)s_0$ is finished for J_i. We have $\epsilon' \le \delta$ because in $\mathcal{S}_t$ at any time $t' > t$ jobs of $\mathcal{J}_0$ are processed at a speed of at least s_0. In these ϵ' time units we cancel the processing of J_i and set the processor speed to 0. Finally, in $[t, t+\delta)$ we process δs_0 units of J_i using speed s_0. These schedule modifications do not increase the idle energy because $\epsilon' \le \delta$. We obtain a feasible schedule that is equal to $\mathcal{S}_{0,3}$ in $[r_{\min}, t+\delta)$.

Next suppose that in $\mathcal{S}_t$ a job J_k is executed throughout $[t, t+\delta)$. If the used speed s is higher than s_0, then we reduce the speed to s_0 and increase w_k by $(s - s_0)\delta$ in the buffer entry (J_k, w_k). If $J_k = J_i$, we are done. If $J_k \ne J_i$, further modifications are required. Again we determine the largest ϵ, where $0 \le \epsilon \le \delta$, such that $\epsilon s_0 \le w_i$ and the next ϵ' time units in $\mathcal{S}_t$ in which a work volume of $(\delta - \epsilon)s_0$ is finished for J_i. Again $0 \le \epsilon' \le \delta$. We modify $\mathcal{S}_t$ by processing J_k using speed s_0 at these ϵ' time units. This can be feasibly done because $\mathcal{S}_{0,3}$ is an *EDF* schedule and hence $d_i \le d_k$. The remaining work of J_k is assigned to B by increasing w_k by $(\delta - \epsilon')s_0$. In $[t, t+\delta)$, we schedule J_i at speed s_0. To this end we take ϵs_0 processing units from the buffer B and hence reduce w_i's value by the corresponding amount. Again the idle energy of the modified schedule has not increased.

Finally, suppose that in $\mathcal{S}_{0,3}$ no job is executed in the interval $[t, t+\delta)$. This implies that among the available jobs, which can be feasibly scheduled in this interval, all jobs are finished. As $\mathcal{S}_t$ is identical to $\mathcal{S}_{0,3}$ in $[r_{\min}, t)$, $\mathcal{S}_t$ cannot process any job in $[t, t+\delta)$ either.

In all the above cases we obtain a (modified) schedule $\mathcal{S}_t$ that is identical to $\mathcal{S}_{0,3}$ in $[r_{\min}, t+\delta)$. The idle energy has not increased. If $t+\delta$ is still within the current interval T_j, we set $t' = t+\delta$. Otherwise we determine the smallest t' with $t' > t+\delta$ such that t' is the beginning of some I_j, where $1 \leq j \leq l$, or of some T_j, where $1 \leq j \leq m$. We set $t' = t$ and the current, modified schedule is the new $\mathcal{S}_t$. Again, if no such t' exits, the transformation is complete.

The above schedule modifications are repeated until the transformation is finally finished. □

The next lemma establishes the first part of inequality (2.3).

Lemma 4. *There holds $E(\mathcal{S}_A) \leq E(\mathcal{S}_0)$.*

Proof. Given our job instance $\mathcal{J} = \{J_1, \dots J_n\}$, let $\mathcal{C}$ be the class of schedules that process $\mathcal{J}_{YDS}$ according to *YDS* and all jobs of $\mathcal{J}_0 = \mathcal{J} \setminus \mathcal{J}_{YDS}$ using speed s_0. Both $\mathcal{S}_A$ and $\mathcal{S}_0$ belong to $\mathcal{C}$. We prove that among the schedules of $\mathcal{C}$, $\mathcal{S}_A$ minimizes energy consumption. The lemma then follows.

Consider any $\mathcal{S} \in \mathcal{C}$. Let $E(\mathcal{J}_{YDS})$ be the energy incurred in processing $\mathcal{J}_{YDS}$ in the intervals $I_1, \dots, I_l$. In these intervals no idle energy is incurred. Any job $J_i \in \mathcal{J}_0$ is processed for v_i/s_0 time units using speed s_0. Let T be the total time for which the processor is in the active state but does not process any jobs in $\mathcal{S}$. Furthermore, let k be the number of wake-up operations performed in $\mathcal{S}$. We have

$$E(\mathcal{S}) = E(\mathcal{J}_{YDS}) + \sum_{J_i \in \mathcal{J}_0} \frac{v_i}{s_0} P(s_0) + TP(0) + kC \tag{2.4}$$

$$= E(\mathcal{J}_{YDS}) - \sum_{j=1}^{l} |I_j| P(0) + \sum_{J_i \in \mathcal{J}_0} \frac{v_i}{s_0} (P(s_0) - P(0)) \tag{2.5}$$

$$+ P(0) \left(\sum_{j=1}^{l} |I_j| + \sum_{J_i \in \mathcal{J}_0} \frac{v_i}{s_0} + T + \frac{kC}{P(0)} \right). \tag{2.6}$$

Let E_1 be the sum given in (2.5). Furthermore, let E_2 be the expression given in the brackets of (2.6). Then E_1 is a fixed overhead incurred by any schedule of $\mathcal{C}$, and E_2 is the cost of a schedule $\mathcal{S}_{BCD}$ that can be obtained from $\mathcal{S}$ for a job instance $\mathcal{J}_{BCD}$ defined as follows. For any $J_i \in \mathcal{J}_0$, we add a job J_i' of processing time $v_i' = v_i/s_0$, release time $r_i' = r_i$ and deadline $d_i' = d_i$ to $\mathcal{J}_{BCD}$. For each interval $I_j = [t_j, t_j')$, where $1 \leq j \leq l$, we add a job $J(I_j)$ of processing time $t_j' - t_j$, release time t_j and deadline t_j' to $\mathcal{J}_{BCD}$. In order to process $\mathcal{J}_{BCD}$ we are given a uniform speed processor. Whenever the processor is in the active state, one cost unit is consumed per time unit. A transition from the sleep state to the active state costs $C/P(0)$.

Given $\mathcal{S}$, we can easily derive a feasible schedule $\mathcal{S}_{BCD}$ for $\mathcal{J}_{BCD}$ that incurs a cost of E_2. In the intervals $I_1, \ldots, I_l$ the *YDS* schedules are replaced by $J(I_1), \ldots, J(I_l)$. The processor speed is set to a uniform speed of say 1. A wake-up operation incurs a cost of $C/P(0)$. Conversely, a feasible schedule $\mathcal{S}_{BCD}$ for $\mathcal{J}_{BCD}$ incurring cost E_2 can be converted into a schedule $\mathcal{S}$ for $\mathcal{J}$ consuming an energy of $E(\mathcal{S})$ as specified in (2.4). In $\mathcal{S}_{BCD}$ we replace the jobs $J(I_1), \ldots, J(I_l)$ by *YDS* schedules for $\mathcal{J}_{YDS}$. At all other times where the processor executes jobs, the speed is set to s_0. A wake-up operation incurs C energy units.

Therefore, the problem of finding a minimum energy schedule in $\mathcal{C}$ is equivalent to computing a minimum cost schedule for $\mathcal{J}_{BCD}$. In Phase 2 algorithm *ALG*(s_0) solves exactly this problem. We conclude that $\mathcal{S}_A$ is a minimum energy schedule in $\mathcal{C}$. □

We proceed to prove the second part of inequality (2.3). Given a schedule $\mathcal{S}$ and a job $J_i \in \mathcal{J}$, let $E_p(\mathcal{S}, J_i)$ denote the processing energy incurred in executing J_i in $\mathcal{S}$. Let $\mathcal{J}_{0,1}$ be the subset of the jobs of $\mathcal{J}_0$ processed at a speed smaller than s_0 in $\mathcal{S}_{OPT}$. Moreover, let $\mathcal{J}_{0,2} = \mathcal{J}_0 \setminus \mathcal{J}_{0,1}$ be the set of remaining jobs of $\mathcal{J}_0$. We present two Lemmas 5 and 6 that hold for our choice of s_0 as specified in Theorem 3.

Lemma 5. *For any $J_i \in \mathcal{J}_{YDS} \cup \mathcal{J}_{0,2}$, there holds $E_p(\mathcal{S}_0, J_i) \leq \frac{4}{3} E_p(\mathcal{S}_{OPT}, J_i)$.*

Proof. Recall that $\mathcal{J}_{YDS}'$, with $\mathcal{J}_{YDS}' \subseteq \mathcal{J}_{YDS}$, is the set of jobs that, using algorithm *YDS*, are processed at a speed higher than s_{crit}. We have chosen $\mathcal{S}_{OPT}$ such that $\mathcal{J}_{YDS}'$ is scheduled according to *YDS*. Moreover, $\mathcal{S}_0$ schedules $\mathcal{J}_{YDS}'$ according to *YDS*. Hence any job $J_i \in \mathcal{J}_{YDS}'$ is processed at the same speed in $\mathcal{S}_0$ and $\mathcal{S}_{OPT}$. We obtain $E_p(\mathcal{S}_0, J_i) = E_p(\mathcal{S}_{OPT}, J_i)$, for any $J_i \in \mathcal{J}_{YDS}'$.

Next consider any $J_i \in \mathcal{J}_{YDS} \setminus \mathcal{J}_{YDS}' \cup \mathcal{J}_{0,2}$. In general, if J_i is processed at speed s, then the incurred processing energy is $\frac{v_i}{s} P(s)$. Since speed s_{crit} minimizes $P(s)/s$, we have that $E_p(\mathcal{S}_{OPT}, J_i) \geq \frac{v_i}{s_{crit}} P(s_{crit})$. In $\mathcal{S}_0$ job J_i is processed at a speed s_i that is at least s_0. Moreover, since $J_i \notin \mathcal{J}_{YDS}'$, we have $s_i \leq s_{crit}$. Due to the fact that s_{crit} is the smallest speed minimizing $P(s)/s$, and by the convexity of the power function, we have $P(s_0)/s_0 \geq P(s_i)/s_i \geq P(s_{crit})/s_{crit}$. By the choice of s_0 there holds $P(s_0)/s_0 = \frac{4}{3} P(s_{crit})/s_{crit}$. Hence $P(s_i)/s_i \leq \frac{4}{3} P(s_{crit})/s_{crit}$ and $E_p(\mathcal{S}_0, J_i) \leq \frac{4}{3} E_p(\mathcal{S}_{OPT}, J_i)$. □

We next turn to the set $\mathcal{J}_{0,1}$. For any $J_i \in \mathcal{J}_{0,1}$, algorithm *Trans* raises the speed to s_0. This speed increase causes idle energy in $\mathcal{S}_{0,1}$ and hence in $\mathcal{S}_0$ because the processor remains in the active state at all the times at which J_i was originally scheduled. For any $J_i \in \mathcal{J}_{0,1}$, let $E_i(J_i)$ be the idle energy incurred. The next lemma implies that, loosely speaking, we can charge $E_i(J_i)$ to $E_p(\mathcal{S}_0, J_i)$.

Lemma 6. *For any $J_i \in \mathcal{J}_{0,1}$, there holds $E_p(\mathcal{S}_0, J_i) + E_i(J_i) \le \frac{4}{3} E_p(\mathcal{S}_{OPT}, J_i)$.*

Proof. Consider an arbitrary job $J_i \in \mathcal{J}_{0,1}$. Let T be the total time for which J_i is processed at a speed s_i, where $s_i < s_0$, in $\mathcal{S}_{OPT}$. Moreover, let T' be the total time for which J_i is executed at speed s_0 in $\mathcal{S}_{0,1}$ and $\mathcal{S}_0$. We have $T' < T$. Choose λ, where $0 \le \lambda \le 1$, such that $T' = \lambda T$. By raising the processor speed to s_0, an idle energy of $(T - T')P(0) = (1-\lambda)TP(0)$ is incurred. In $\mathcal{S}_0$ the processing energy for J_i is $T'P(s_0) = \lambda T P(s_0)$. Hence

$$\begin{aligned} E_p(\mathcal{S}_0, J_i) + E_i(J_i) &= \lambda T P(s_0) + (1-\lambda) T P(0) \\ &= T(P(0) + \lambda(P(s_0) - P(0))). \end{aligned}$$

The processing volume of J_i is $T's_0 = \lambda T s_0$ and hence J_i is executed at speed λs_0 in $\mathcal{S}_{OPT}$. We obtain $E_p(\mathcal{S}_{OPT}, J_i) = TP(\lambda s_0)$. We substitute λs_0 by s, which implies $\lambda = s/s_0$. Let

$$R(s) = \frac{E_p(\mathcal{S}_0, J_i) + E_i(J_i)}{E_p(\mathcal{S}_{OPT}, J_i)} = \frac{P(0) + \frac{s}{s_0}(P(s_0) - P(0))}{P(s)}. \tag{2.7}$$

In order to establish the lemma we have to show that the ratio $R(s)$ is upper bounded by $4/3$, for all s with $0 \le s \le s_0$. Consider the ratio in (2.7). The numerator is a line, which we denote by $f(s)$, passing through $(0, P(0))$ and $(s_0, P(s_0))$. We have to compare $f(s)$ to $P(s)$ in the interval $[0, s_0]$, see Figure 2.8.

Let $g(s)$ be the line passing through $(s_0, P(s_0))$ and $(s_{crit}, P(s_{crit}))$. We will use this line to lower bound $P(s)$. Line $g(s)$ has a slope of $(P(s_{crit}) - P(s_0))/(s_{crit} - s_0)$, which is smaller than $P(s_{crit})/s_{crit}$. This holds true because $(P(s_{crit}) - P(s_0))/(s_{crit} - s_0) < P(s_{crit})/s_{crit}$ is equivalent to $P(s_{crit})/s_{crit} < P(s_0)/s_0$. The latter inequality is satisfied as s_{crit} is the smallest value minimizing $P(s)/s$. The line connecting $(0,0)$ and $(s_{crit}, P(s_{crit}))$ has a slope of $P(s_{crit})/s_{crit}$. Since g has a smaller slope, we have $g(0) \ge 0$.

Moreover, we have $f(s_0) = P(s_0) = g(s_0)$. The power function P is convex and hence the slope of f cannot be greater than the slope of g. Since $f(0) = P(0)$, it follows $g(0) \le P(0)$. We conclude $0 \le g(0) \le P(0)$. Choose a constant a, where $0 \le a \le 1$, such that $g(0) = aP(0)$. Then $g(s)$, which passes through $(s_{crit}, P(s_{crit}))$, is defined as follows.

$$g(s) = aP(0) + \frac{P(s_{crit}) - aP(0)}{s_{crit}} s.$$

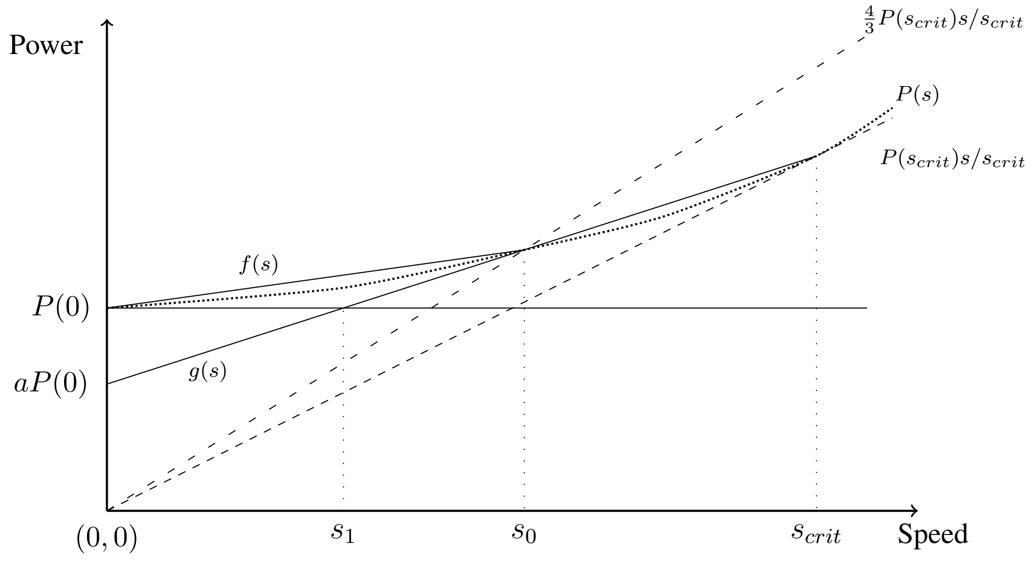


Figure 2.8: Lines $f(s)$ and $g(s)$.

There holds $g(s_0) = P(s_0)$. By our choice of s_0, we have $P(s_0) = \frac{4}{3}P(s_{crit})s_0/s_{crit}$. We solve the equation $g(s_0) = \frac{4}{3}P(s_{crit})s_0/s_{crit}$ for s_0. Moreover, let s_1 be the value satisfying $g(s_1) = P(0)$. Then

$$s_0 = \frac{3aP(0)s_{crit}}{P(s_{crit}) + 3aP(0)}, \quad \text{and} \quad s_1 = \frac{(1-a)P(0)s_{crit}}{P(s_{crit}) - aP(0)}.$$

Line g passes through $(s_0, P(s_0))$ and $(s_{crit}, P(s_{crit}))$. Hence, by convexity of P, we have $g(s) \leq P(s)$, for all s with $0 \leq s \leq s_0$. Moreover, $P(s) \geq P(0)$, for all $s \geq 0$. Let $P'(s)$ be the following function:

$$P'(s) = \begin{cases} P(0), & 0 \leq s < s_1, \\ g(s), & s_1 \leq s \leq s_0. \end{cases}$$

Then $P'(s) \leq P(s)$ for all $0 \leq s \leq s_0$, and instead of upper bounding $R(s)$ by $4/3$ we will show that $R'(s) = (P(0) + \frac{s}{s_0}(P(s_0) - P(0)))/P'(s)$ is upper bounded by $4/3$, for all $0 \leq s \leq s_0$. Line $f(s)$ is again the numerator of $R'(s)$. The function f is increasing and $f(s) \geq P(0)$, for all $s \geq 0$. In the interval $[0, s_1)$, we have $P'(s) = P(0)$. Hence for any $s \in [0, s_1)$, there holds $R'(s) \leq f(s_1)/P(0) = f(s_1)/g(s_1)$. In the interval $[s_1, s_0]$, we have $R'(s) = f(s)/g(s)$. Both functions f and g are increasing. There holds $f(s_1) \geq P(0) = g(s_1)$ and $f(s_0) = P(s_0) = g(s_0)$. It follows $R'(s) \geq R'(s')$, for all $s_1 \leq s < s' \leq s_0$. Hence $R'(s)$ is maximized for $s = s_1$ and it suffices to show that $R'(s_1) =$

$f(s_1)/g(s_1) = f(s_1)/P(0)$ is upper bounded by $4/3$. We have

$$\begin{aligned} f(s_1) &= P(0) + \tfrac{s_1}{s_0}(P(s_0) - P(0)) \\ &= P(0) + \left(\tfrac{4}{3}\tfrac{P(s_{crit})}{s_{crit}} - \tfrac{P(0)}{s_0}\right) s_1 \\ &= P(0) + \left(\tfrac{4}{3}\tfrac{P(s_{crit})}{s_{crit}} - \tfrac{P(s_{crit})+3aP(0)}{3as_{crit}}\right) \tfrac{(1-a)P(0)s_{crit}}{P(s_{crit})-aP(0)} \\ &= P(0) + \left(\tfrac{(4a-1)P(s_{crit})-3aP(0)}{P(s_{crit})-aP(0)}\right) \tfrac{1-a}{3a} P(0). \end{aligned}$$

The second equation holds because $P(s_0)/s_0 = \frac{4}{3}P(s_{crit})/s_{crit}$. The third equation is obtained by plugging in the values of s_0 and s_1. Hence

$$R'(s_1) = 1 + \frac{1}{3}\left(\frac{(4a-1)P(s_{crit}) - 3aP(0)}{P(s_{crit}) - aP(0)}\right)\frac{1-a}{a},$$

and it remains to show that the last term in the above expression is upper bounded by $1/3$. This is equivalent to proving $-4(a-1/2)^2P(s_{crit}) + (4a^2-3a)P(0) \leq 0$. We have $-4(a-1/2)^2 \leq 0$, for any $0 \leq a \leq 1$, and $P(s_{crit}) \geq P(0) > 0$. We conclude, as desired,

$$\begin{aligned} &-4(a-1/2)^2P(s_{crit}) + (4a^2-3a)P(0) \\ \leq &-4(a-1/2)^2P(0) + (4a^2-3a)P(0) = (a-1)P(0) \leq 0. \end{aligned}$$ □

We now prove the second part of inequality (2.3). Consider again the schedule $\mathcal{S}_{0,1}$ obtained after Step 1 of *Trans*. Compared to $\mathcal{S}_{OPT}$, the idle energy increases by $\sum_{J_i\in\mathcal{J}_{0,1}} E_i(J_i)$. Hence $E_i(\mathcal{S}_{0,1}) = E_i(\mathcal{S}_{OPT}) + \sum_{J_i\in\mathcal{J}_{0,1}} E_i(J_i)$ and by Lemma 3 $E_i(\mathcal{S}_0) \leq E_i(\mathcal{S}_{OPT}) + \sum_{J_i\in\mathcal{J}_{0,1}} E_i(J_i)$. We conclude

$$\begin{aligned} E(\mathcal{S}_0) &= E_p(\mathcal{S}_0) + E_i(\mathcal{S}_0) \\ &= \sum_{J_i\in\mathcal{J}_{YDS}\cup\mathcal{J}_{0,2}} E_p(\mathcal{S}_0, J_i) + \sum_{J_i\in\mathcal{J}_{0,1}} E_p(\mathcal{S}_0, J_i) + E_i(\mathcal{S}_0) \\ &\leq \sum_{J_i\in\mathcal{J}_{YDS}\cup\mathcal{J}_{0,2}} \tfrac{4}{3}E_p(\mathcal{S}_{OPT}, J_i) + \sum_{J_i\in\mathcal{J}_{0,1}} (E_p(\mathcal{S}_0, J_i) + E_i(J_i)) + E_i(\mathcal{S}_{OPT}) \\ &\leq \sum_{J_i\in\mathcal{J}_{YDS}\cup\mathcal{J}_{0,2}} \tfrac{4}{3}E_p(\mathcal{S}_{OPT}, J_i) + \sum_{J_i\in\mathcal{J}_{0,1}} \tfrac{4}{3}E_p(\mathcal{S}_{OPT}, J_i) + E_i(\mathcal{S}_{OPT}) \\ &= \tfrac{4}{3}E_p(\mathcal{S}_{OPT}) + E_i(\mathcal{S}_{OPT}) \\ &\leq \tfrac{4}{3}E(\mathcal{S}_{OPT}). \end{aligned}$$

The first inequality follows from Lemma 5 and our bound on $E_i(\mathcal{S}_0)$. The second one follows from Lemma 6.

2.3 Power functions $P(s) = \beta s^\alpha + \gamma$

We develop an improved approximation guarantee for the family of power functions $P(s) = \beta s^\alpha + \gamma$, where $\alpha > 1$ and $\beta, \gamma > 0$ are constants. The critical speed, i.e., the speed minimizing $P(s)/s$, is $s_{crit} = \sqrt[\alpha]{\gamma/(\beta(\alpha-1))}$. Let $s_\alpha = \frac{c}{c+1} s_{crit}$, where $c = 117/20 = 5.85$, and *ALG*(s_α) be the algorithm obtained from *ALG*(s_0) be setting s_0 to s_α.

Theorem 4. *ALG*(s_α) *achieves an approximation factor of* $(c+1)/c = 137/117 < 1.171$.

The proof of Theorem 4 proceeds along the same lines as the proof of Theorem 3 and we have to replace Lemmas 5 and 6.

Lemma 7. *For any* $J_i \in \mathcal{J}_{YDS} \cup \mathcal{J}_{0,2}$*, there holds* $E_p(\mathcal{S}_0, J_i) \le \frac{c+1}{c} E_p(\mathcal{S}_{OPT}, J_i)$.

Proof. For any $J_i \in \mathcal{J}'_{YDS}$ there holds $E_p(\mathcal{S}_0, J_i) = E_p(\mathcal{S}_{OPT}, J_i)$. Recall that $s_\alpha = c/(c+1)s_{crit}$. This implies $P(s_\alpha)/s_\alpha = \frac{c+1}{c} P(\frac{c}{c+1} s_{crit})/s_{crit} \le \frac{c+1}{c} P(s_{crit})/s_{crit}$. The inequality holds because $P(s)$ is nondecreasing. Hence, for any $J_i \in \mathcal{J}_{YDS} \setminus \mathcal{J}'_{YDS} \cup \mathcal{J}_{0,2}$, we have

$$E_p(\mathcal{S}_0, J_i) \le v_i P(s_\alpha)/s_\alpha \le \frac{c+1}{c} v_i P(s_{crit})/s_{crit} \le \frac{c+1}{c} E_p(\mathcal{S}_{OPT}, J_i). \quad \square$$

We need the following technical lemma. The proof requires the Lambert W function whose defining equation, for any number x, is $x = W(x)e^{W(x)}$. The function is double-valued in the interval $(-1/e, 0)$, and $W_{-1}(x)$ and $W_0(x)$ denote the lower and upper branches, respectively.

Lemma 8. *The inequality*

$$\frac{\lambda\left(\frac{c}{c+1}\right)^\alpha + \alpha - 1}{\lambda^\alpha\left(\frac{c}{c+1}\right)^\alpha + \alpha - 1} \le \frac{c+1}{c},$$

holds for $c = 117/20$*, all* $\alpha > 1$ *and all* $0 \le \lambda \le 1$.

Proof. For simplicity set $x = \frac{c+1}{c}$. We then have to show $\frac{\lambda\left(\frac{1}{x}\right)^\alpha + \alpha - 1}{\lambda^\alpha\left(\frac{1}{x}\right)^\alpha + \alpha - 1} \le x$, which is equivalent to

$$\begin{aligned} & \lambda(1/x)^\alpha + (\alpha-1) \le x\lambda^\alpha(1/x)^\alpha + x(\alpha-1) \\ \Leftrightarrow \quad & (x\lambda^\alpha - \lambda)(1/x)^\alpha + (x-1)(\alpha-1) \ge 0. \end{aligned}$$

Set $f(\lambda) = (x\lambda^\alpha - \lambda)(1/x)^\alpha + (x-1)(\alpha-1)$. It follows that $f'(\lambda) = \alpha\lambda^{\alpha-1}(1/x)^{\alpha-1} - (1/x)^\alpha$ and $f''(\lambda) = \alpha(\alpha-1)\lambda^{\alpha-2}(1/x)^{\alpha-1}$, which is non-negative. This implies that f is convex and gets minimized for λ satisfying $\alpha\lambda^{\alpha-1} = \frac{1}{x}$. Hence

$$\lambda = \left(\frac{1}{x\alpha}\right)^{\frac{1}{\alpha-1}}.$$

Thus, it is sufficient to show that

$$\left[x\left(\frac{1}{x\alpha}\right)^{\alpha/(\alpha-1)} - \left(\frac{1}{x\alpha}\right)^{1/(\alpha-1)}\right]\left(\frac{1}{x}\right)^\alpha + (x-1)(\alpha-1) \geq 0,$$

which is equivalent to

$$\begin{aligned} & \left(\frac{1}{x\alpha}\right)^{\frac{1}{\alpha-1}}\left(\frac{1}{\alpha}-1\right)\left(\frac{1}{x}\right)^\alpha + (x-1)(\alpha-1) \geq 0 \\ \Leftrightarrow \quad & \left(\frac{1}{x\alpha}\right)^{\frac{1}{\alpha-1}}\frac{1}{\alpha}\left(\frac{1}{x}\right)^\alpha \leq x-1. \end{aligned}$$

By setting x back to $\frac{c+1}{c}$, we obtain

$$\begin{array}{rrcl} & \left(\frac{c}{(c+1)\alpha}\right)^{\frac{1}{\alpha-1}}\left(\frac{c}{c+1}\right)^\alpha & \leq & \frac{1}{c}\cdot\alpha \\ \Leftrightarrow & c^{\alpha+1+\frac{1}{\alpha-1}} & \leq & \alpha^{1+\frac{1}{\alpha-1}}\cdot(c+1)^{\alpha+\frac{1}{\alpha-1}} \\ \Leftrightarrow & c^{\alpha^2} & \leq & \alpha^\alpha\cdot(c+1)^{\alpha^2-\alpha+1}. \end{array}$$

The last step follows by raising to the power of $\alpha-1$. By taking the logarithm and dividing by α^2 we have

$$\ln(c+1) - \ln(c) + \frac{1-\alpha}{\alpha^2}\ln(c+1) + \frac{1}{\alpha}\ln(\alpha) \geq 0.$$

Let $g(\alpha) = \ln(c+1) - \ln(c) + \frac{1-\alpha}{\alpha^2}\ln(c+1) + \frac{1}{\alpha}\ln(\alpha)$. We wish to show that $g(\alpha) \geq 0$. It suffices to show that $g(\alpha) \geq 0$ for the extrema of g in $(1,+\infty)$. These are attained when $\alpha \to 1^+$, $\alpha \to +\infty$, and at the points where $g'(\alpha) = 0$. For the first two, we have

$$\lim_{\alpha\to 1^+} g(a) = \lim_{\alpha\to+\infty} g(a) = \ln(c+1) - \ln(c),$$

which is strictly positive for any $c > 0$. We next determine the roots of $g'(\alpha) = 0$. There holds

$$g'(\alpha) = \frac{\alpha-2}{\alpha^3}\ln(c+1) - \frac{\ln(\alpha)}{\alpha^2} + \frac{1}{\alpha^2}.$$

Thus

$$\begin{aligned} g'(\alpha) &= 0 \\ \Leftrightarrow \quad (\alpha-2)\ln(c+1) - \alpha\ln(\alpha) + \alpha &= 0 \\ \Leftrightarrow \quad \ln\left(\frac{(c+1)^{\alpha-2}}{\alpha^\alpha}\right) &= -\alpha \\ \Leftrightarrow \quad (c+1)^\alpha e^\alpha &= \alpha^\alpha (c+1)^2 \\ \Leftrightarrow \quad \frac{1}{\alpha}(c+1)e &= (c+1)^{\frac{2}{\alpha}}. \end{aligned}$$

By setting $t = -\frac{2}{\alpha}$ we obtain

$$\begin{aligned} -\frac{c+1}{2}et &= (c+1)^{-t} \\ \Leftrightarrow \quad t(c+1)^t &= -\frac{2}{(c+1)e}, \end{aligned}$$

which gives

$$t = \frac{W\left(-\frac{2\ln(c+1)}{(c+1)e}\right)}{\ln(c+1)},$$

giving the following roots for $g'(\alpha) = 0$:

$$\alpha_1 = -\frac{2\ln(c+1)}{W_0\left(-\frac{2\ln(c+1)}{(c+1)e}\right)}, \quad \text{and} \quad \alpha_2 = -\frac{2\ln(c+1)}{W_{-1}\left(-\frac{2\ln(c+1)}{(c+1)e}\right)}.$$

By evaluating g at α_1 and α_2, with $c = 117/20$, we get 0.2186875389 and 0.0000352487 respectively. Since both values are non-negative the proof is complete. □

Lemma 9. *For any $J_i \in \mathcal{J}_{0,1}$, there holds $E_p(\mathcal{S}_0, J_i) + E_i(J_i) \leq \frac{c+1}{c} E_p(\mathcal{S}_{OPT}, J_i)$.*

Proof. Let J_i be any job in $\mathcal{J}_{0,1}$. Let T be the total time for which J_i is executed using a speed of $s_i < s_\alpha$ in $\mathcal{S}_{OPT}$. Let $T' < T$ be the total time for which the job is processed at speed s_α in $\mathcal{S}_{0,1}$ and $\mathcal{S}_0$. Choose λ, with $0 \leq \lambda \leq 1$, such that $T' = \lambda T$. We have $s_\alpha = \frac{c}{c+1} s_{crit}$ and $s_{crit} = \sqrt[\alpha]{\gamma/(\beta(\alpha-1))}$. Therefore, $P(s_\alpha) = (\frac{c}{c+1})^\alpha \gamma/(\alpha-1) + \gamma$ and

$$\begin{aligned} E_p(\mathcal{S}_0, J_i) + E_i(J_i) &\leq \lambda T P(s_\alpha) + (1-\lambda) T P(0) \\ &= T(\lambda(\frac{c}{c+1})^\alpha \gamma/(\alpha-1) + \gamma). \end{aligned}$$

In $\mathcal{S}_{OPT}$ job J_i is processed at speed λs_α and thus

$$E_p(\mathcal{S}_{OPT}, J_i) = TP(\lambda s_\alpha) = T(\lambda^\alpha(\frac{c}{c+1})^\alpha \gamma/(\alpha-1)+\gamma).$$

By the above Lemma 8, the ratio

$$\frac{E_p(\mathcal{S}_0, J_i) + E_i(J_i)}{E_p(\mathcal{S}_{OPT}, J_i)} \le \frac{\lambda(\frac{c}{c+1})^\alpha \gamma/(\alpha-1)+\gamma}{\lambda^\alpha(\frac{c}{c+1})^\alpha \gamma/(\alpha-1)+\gamma} = \frac{\lambda(\frac{c}{c+1})^\alpha + \alpha - 1}{\lambda^\alpha(\frac{c}{c+1})^\alpha + \alpha - 1}$$

is upper bounded by $(c+1)/c$. □

Using Lemmas 7 and 9 we can show $E(\mathcal{S}_0) \le \frac{c+1}{c} E(\mathcal{S}_{OPT})$. The calculation is similar to that presented at the end of Section 2.2.

2.4 Revisiting s_{crit}-schedules

In this section we show that the algorithmic framework presented in Section 2.2 yields the best possible approximation guarantees that can be achieved by algorithms constructing s_{crit}-schedules, both for general convex power functions and for the family of functions $P(s) = \beta s^\alpha + \gamma$. Let *ALG*$(s_{crit})$ be the algorithm *ALG*(s_0), where s_0 is set to s_{crit}.

Theorem 5. *ALG*(s_{crit}) *achieves an approximation factor of* 2*, for general convex power functions.*

Theorem 6. *ALG*(s_{crit}) *achieves an approximation factor of*

$$\frac{eW_{-1}(-e^{-1-\frac{1}{e}})}{eW_{-1}(-e^{-1-\frac{1}{e}})+1},$$

for power functions $P(s) = \beta s^\alpha + \gamma$*, where* $\alpha > 1$ *and* $\beta, \gamma > 0$.

The ratio given in Theorem 6 is smaller than 1.211. In order to prove Theorems 5 and 6 we have to replace Lemmas 5 and 6. For $s_0 = s_{crit}$, the set $\mathcal{J}_{YDS} \setminus \mathcal{J}'_{YDS}$ of jobs scheduled according to *YDS* but at speeds of at most s_{crit} is empty. Hence $\mathcal{J}_{YDS} = \mathcal{J}'_{YDS}$. Let $\mathcal{S}_{OPT}$ denote again an optimal schedule in which the jobs of $\mathcal{J}_{YDS}$ are scheduled according to *YDS*. Schedule $\mathcal{S}_0$ is obtained from $\mathcal{S}_{OPT}$ by applying *Trans* and satisfies the additional property that all jobs of $\mathcal{J}_0$ are executed at speed s_{crit}. Any job $J_i \in \mathcal{J}_{YDS}$ is processed at the same speed in $\mathcal{S}_0$ and $\mathcal{S}_{OPT}$. Hence $E_p(\mathcal{S}_0, J_i) = E_p(\mathcal{S}_{OPT}, J_i)$, for any $J_i \in \mathcal{J}_{YDS}$, and this fact replaces Lemma 5. For the proof of Theorem 5 we show the following lemma.

Lemma 10. *For any $J_i \in \mathcal{J}_0$, there holds $E_p(\mathcal{S}_0, J_i) + E_i(J_i) \leq 2E_p(\mathcal{S}_{OPT}, J_i)$.*

Proof. Let $J_i \in \mathcal{J}_0$ be an arbitrary job. As for the processing energy, $E_p(\mathcal{S}_0, J_i) = v_i P(s_{crit})/s_{crit} \leq E_p(\mathcal{S}_{OPT}, J_i)$ because s_{crit} is the speed minimizing $P(s)/s$. Suppose that J_i is processed for T time units in $\mathcal{S}_{OPT}$. If J_i's speed is raised to s_{crit} in Step 1 of *Trans*, the extra idle energy incurred cannot be higher than $TP(0)$, which in turn is a lower bound on the processing energy incurred for J_i in $\mathcal{S}_{OPT}$. Hence $E_i(J_i) \leq E_p(\mathcal{S}_{OPT}, J_i)$, and the lemma follows. $\square$

Using Lemma 10 we can prove the desired inequality $E(\mathcal{S}_0) \leq 2E(\mathcal{S}_{OPT})$; the calculation is similar to that presented at the end of Section 2.2. This concludes the proof of Theorem 5. For the proof of Theorem 6 we need another technical lemma.

Lemma 11. *The inequality*

$$\frac{\lambda + \alpha - 1}{\lambda^\alpha + \alpha - 1} \leq \frac{eW_{-1}(-e^{-1-1/e})}{eW_{-1}(-e^{-1-1/e}) + 1}$$

holds for all $\alpha > 1$ and $0 \leq \lambda \leq 1$. Furthermore there exist $\alpha = \alpha' > 1$ and $\lambda = \lambda' \in [0, 1]$ such that equality holds.

Proof. Assume that $(\lambda + \alpha - 1)/(\lambda^\alpha + \alpha - 1) \leq x$. We will show that the smallest possible x satisfying this inequality, is $eW(-e^{-1-1/e})/(eW(-e^{-1-1/e}) + 1)$. Inequality $(\lambda + \alpha - 1)/(\lambda^\alpha + \alpha - 1) \leq x$ is equivalent to

$$x\lambda^\alpha + x(\alpha - 1) - \lambda - (\alpha - 1) \geq 0.$$

Setting $f(\lambda) = x\lambda^\alpha - \lambda + (x - 1)(\alpha - 1)$, we have $f'(\lambda) = x\alpha\lambda^{\alpha-1} - 1$ and $f''(\lambda) \geq 0$. It follows that $f(\lambda)$ is minimized for $\lambda = (\frac{1}{x\alpha})^{\frac{1}{\alpha-1}}$, and it suffices to find the minimal x so that

$$x\left(\frac{1}{x\alpha}\right)^{\frac{\alpha}{\alpha-1}} - \left(\frac{1}{x\alpha}\right)^{\frac{1}{\alpha-1}} + (x - 1)(\alpha - 1) \geq 0$$

holds for every $\alpha > 1$. The latter inequality is equivalent to

$$\begin{aligned}
\left(\frac{1}{\alpha} - 1\right)\left(\frac{1}{x\alpha}\right)^{\frac{1}{\alpha-1}} + (x - 1)(\alpha - 1) &\geq 0 \\
\Leftrightarrow \quad (\alpha - 1)(x - 1 - \frac{\left(\frac{1}{x\alpha}\right)^{\frac{1}{\alpha-1}}}{\alpha}) &\geq 0 \\
\Leftrightarrow \quad x - 1 - \frac{\left(\frac{1}{x\alpha}\right)^{\frac{1}{\alpha-1}}}{\alpha} &\geq 0 \\
\Leftrightarrow \quad x\alpha^\alpha(x - 1)^{\alpha-1} &\geq 1.
\end{aligned}$$

Substituting x by $\frac{c+1}{c}$, the last inequality becomes $(c+1)\alpha^\alpha \geq c^\alpha$, and we seek the largest possible c so that it is satisfied. We have

$$(c+1)\alpha^\alpha \geq c^\alpha$$

or equivalently,

$$\ln(c+1) + \alpha(\ln\alpha - \ln c) \geq 0.$$

Let $g(\alpha) = \ln(c+1) + \alpha(\ln\alpha - \ln c)$. By derivating, we obtain $g'(\alpha) = \ln\alpha - \ln c + 1$ and $g''(\alpha) = 1/\alpha \geq 0$, which implies that g is minimized for $\alpha = c/e$. We therefore seek the largest possible c such that $\ln(c+1) \geq c/e$ holds.

For the largest possible c, equality holds, i.e., $\ln(c+1) = c/e$. By setting $c = -et - 1$ the equality becomes $te^t = -e^{-1-1/e}$. It follows that $t = W(-e^{-1-1/e})$, and $c = -et - 1 = -eW_{-1}(-e^{-1-1/e}) - 1$, which leads to

$$x = \frac{c+1}{c} = \frac{eW_{-1}(-e^{-1-1/e})}{eW_{-1}(-e^{-1-1/e}) + 1},$$

concluding the proof for the first statement of the lemma. Note that we are only interested in the lower branch of the W function, since $W_0(-e^{-1-\frac{1}{e}}) = W_0(-\frac{1}{e}e^{-\frac{1}{e}}) = -\frac{1}{e}$ resulting in $c = 0$.

For the second statement, it is sufficient to show that the extrema α' and λ', by which we substituted α and λ in the above analysis, are greater than 1 and in the range $[0,1]$, respectively. For α we have

$$\alpha' = \frac{c}{e} = \frac{-eW_{-1}(-e^{-1-1/e}) - 1}{e} > 1.$$

The last step holds because W_{-1} is a monotonically decreasing function, which implies that $W_{-1}(-e^{-1-\frac{1}{e}}) \leq W_{-1}((-1-\frac{1}{e})^{-1-\frac{1}{e}})$ and in turn $W_{-1}(-e^{-1-\frac{1}{e}}) < -1 - \frac{1}{e}$. As for λ, we first prove that $\lambda' \leq 1$. It suffices to show that $x\alpha \geq 1$ and hence that $x \geq 1$. We thus have to show

$$\frac{eW_{-1}(-e^{-1-1/e})}{eW_{-1}(-e^{-1-1/e}) + 1} \geq 1. \tag{2.8}$$

The last inequality is satisfied: We already observed that $W_{-1}(-e^{-1-\frac{1}{e}}) < -1 - \frac{1}{e}$ which is less than $-1/e$, and (2.8) becomes $eW_{-1}(-e^{-1-1/e}) \leq eW_{-1}(-e^{-1-1/e}) + 1$. It remains to prove that $\lambda' \geq 0$, which is again equivalent to showing that $W_{-1}(-e^{-1-\frac{1}{e}}) \leq -\frac{1}{e}$. □

Lemma 12. *For any $J_i \in \mathcal{J}_0$, there holds $E_p(\mathcal{S}_0, J_i) + E_i(J_i) \leq cE_p(\mathcal{S}_{OPT}, J_i)$, where $c = eW_{-1}(-e^{-1-1/e})/(eW_{-1}(-e^{-1-1/e}) + 1)$.*

Proof. Let $J_i \in \mathcal{J}_0$ be an arbitrary job. If J_i is processed at speed s_{crit} in $\mathcal{S}_{OPT}$, there is nothing to show. So assume that J_i is processed at a speed smaller than s_{crit}. Let T and T' be the total times for which J_i is executed in $\mathcal{S}_{OPT}$ and $\mathcal{S}_0$, respectively. Choose λ, with $0 \leq \lambda \leq 1$, such that $T' = \lambda T$. We have $s_{crit} = \sqrt[\alpha]{\gamma/(\beta(\alpha-1))}$ and $P(s_{crit}) = \gamma/(\alpha-1) + \gamma$. Hence $E_p(\mathcal{S}_0, J_i) + E_i(J_i) \leq \lambda T P(s_{crit}) + (1-\lambda) T P(0) = T(\lambda\gamma/(\alpha-1) + \gamma)$. In $\mathcal{S}_{OPT}$ job J_i is processed at speed λs_{crit} and thus $E_p(\mathcal{S}_{OPT}, J_i) = TP(\lambda s_{crit}) = T(\lambda^\alpha \gamma/(\alpha-1) + \gamma)$. By Lemma 11, as desired,

$$\frac{E_p(\mathcal{S}_0, J_i) + E_i(J_i)}{E_p(\mathcal{S}_{OPT}, J_i)} \leq \frac{\lambda\gamma/(\alpha-1) + \gamma}{\lambda^\alpha\gamma/(\alpha-1) + \gamma}$$
$$= \frac{\lambda + \alpha - 1}{\lambda^\alpha + \alpha - 1} \leq \frac{eW_{-1}(-e^{-1-1/e})}{eW_{-1}(-e^{-1-1/e}) + 1}. \qquad \square$$

Using Lemma 12 we can show $E(\mathcal{S}_0) \leq cE(\mathcal{S}_{OPT})$, where $c = eW_{-1}(-e^{-1-1/e})/(eW_{-1}(-e^{-1-1/e}) + 1)$. The calculation is similar to the one presented at the end of Section 2.2. This establishes Theorem 6. For power functions $P(s) = \beta s^\alpha + \gamma$, we prove a matching lower bound on the performance of s_{crit}-schedules.

Theorem 7. *Let A be an algorithm that computes s_{crit}-schedules for any job instance. Then A does not achieve an approximation factor smaller than $eW_{-1}(-e^{-1-\frac{1}{e}})/(eW_{-1}(-e^{-1-\frac{1}{e}}) + 1)$, for power functions $P(s) = \beta s^\alpha + \gamma$, where $\alpha > 1$ and $\beta, \gamma > 0$.*

Proof. Let ϵ, $0 < \epsilon < 1$, be a constant. We show that A cannot achieve an approximation factor smaller than

$$\frac{eW_{-1}(-e^{-1-\frac{1}{e}})}{eW_{-1}(-e^{-1-\frac{1}{e}}) + 1} - \epsilon.$$

Fix a power function $P(s) = \beta s^{\alpha'} + \gamma$, where α' is defined as in Lemma 11. Then $s_{crit} = \sqrt[\alpha']{\gamma/(\beta(\alpha'-1))}$. We specify a job sequence that is similar to the one in the proof of Theorem 2. Let again $L > 0$ be an arbitrary constant. We define three jobs J_1, J_2 and J_3: Jobs J_1 and J_3 both have a processing volume of $v_1 = v_3 = \delta L s_{crit}$ and can be executed in intervals $I_1 = [0, \delta L)$ and $I_3 = [(1+\delta)L, (1+2\delta)L)$, respectively. Job J_2 has a processing volume of $v_2 = \lambda' L s_{crit}$, where λ' is as defined in Lemma 11, and can be executed in $I_2 = [\delta L, (1+\delta)L)$. The energy consumed by a wake-up operation is $C = LP(0) = \gamma L$.

As in the proof of Theorem 2 we first assume that the processor is in the active state at time 0. We analyze the energy of algorithm A and an optimal solution. We assume that A processes all the three jobs at speed s_{crit}. If a job is processed at a higher speed, we can reduce its speed to s_{crit}. This reduction of speed, reduces the processing energy of the job and does not increase the idle energy of the schedule. Hence jobs J_1 and J_3 consume an energy of $\delta LP(s_{crit})$ each. Job J_2 is processed for $v_2/s_{crit} = \lambda' L$ time units in I_2, resulting in an energy consumption of $\lambda' LP(s_{crit}) = \lambda' L(\gamma/(\alpha'-1)+\gamma)$. During the $(1-\lambda')L$ time units remaining in I_2 the processor is idle. Since $C > (1-\lambda')LP(0)$ the processor should stay in the active state for this amount of time. It follows that the energy consumption of A is at least

$$2\delta LP(s_{crit}) + \lambda' LP(s_{crit}) + (1-\lambda')LP(0) > L\left(\frac{\lambda'\gamma}{\alpha'-1}+\gamma\right).$$

An optimal solution will also process J_1 and J_3 at speed s_{crit}. However, J_2 can be executed during the whole interval I_2 at speed $\lambda' s_{crit}$, yielding an energy consumption of $LP(\lambda' s_{crit})$. It follows that the energy consumption of an optimal solution is upper bounded by

$$2\delta LP(s_{crit}) + LP(\lambda' s_{crit}) = L\left(2\delta P(s_{crit}) + (\lambda')^{\alpha'}\frac{\gamma}{\alpha'-1}+\gamma\right).$$

Now assume that the processor is in the sleep state at time 0. We repeat the above job sequence k times, where the value of k will be determined later. For each repetition i, $1 \le i \le k$, three jobs are introduced. The job J_{i1}, J_{i2} and J_{i3} have processing volumes of $v_{ij} = v_j$, for $1 \le j \le 3$, and respective execution intervals $[t_i, t_i+\delta L)$, $[t_i+\delta L, t_i+(1+\delta)L)$ and $[t_i+(1+\delta)L, t_i+(1+2\delta)L)$, where $t_i = (i-1)(1+2\delta)L$. For this job sequence, the ratio of the energy consumed by A to that of an optimal solution is at least

$$\frac{kL\left(\lambda'\frac{\gamma}{\alpha'-1}+\gamma\right)}{C + kL\left(2\delta P(s_{crit}) + (\lambda')^{\alpha'}\frac{\gamma}{\alpha'-1}+\gamma\right)}.$$

Set $X = \frac{\lambda'}{\alpha'-1}+1$ and $Y = \frac{(\lambda')^{\alpha'}}{\alpha'-1}+1$. Moreover, set $\epsilon' = \epsilon Y^2/X$. With these settings, let $\delta = \epsilon'\gamma/(4P(s_{crit}))$ and $k = \lceil 2/\epsilon' \rceil$. Then $2\delta P(s_{crit}) \le \epsilon'\gamma/2$ and $C = kC/k \le kC\epsilon'/2 = kL\gamma\epsilon'/2$. Hence the above ratio is at least

$$\frac{\frac{\lambda'}{\alpha'-1}+1}{\frac{(\lambda')^{\alpha'}}{\alpha'-1}+1+\epsilon'} \ge \frac{\frac{\lambda'}{\alpha'-1}+1}{\frac{(\lambda')^{\alpha'}}{\alpha'-1}+1} - \epsilon = \frac{\lambda'+\alpha'-1}{(\lambda')^{\alpha'}+\alpha'-1} - \epsilon.$$

The first inequality holds because $X/(Y+\epsilon') \ge X/Y - \epsilon$ by our choice of X, Y and ϵ'. The theorem now follows from Lemma 11. $\square$

Chapter 3

Multiprocessor Speed Scaling

Almost all of the previous studies on dynamic speed scaling assume that a single variable-speed processor is given. However, energy conservation and speed scaling techniques are equally interesting in multi-processor environments. Multicore platforms will be the dominant processor architectures in the future. Nowadays many PCs and laptops are already equipped with dual-core and quad-core designs. Moreover, computer clusters and server farms, usually consisting of many high-speed processors, represent parallel multi-processor systems that have been used successfully in academia and enterprises for many years. Power dissipation has become a major concern in these environments. Additionally, in research, multi-processor systems have always been investigated extensively, in particular as far as scheduling and resource management problems are concerned.

In this chapter we study dynamic speed scaling in multi-processor environments. We adopt the framework by Yao et al. [55], but assume that m parallel processors are given. Recall that in the problem introduced by Yao et al., we are given a sequence $\sigma = J_1, \ldots, J_n$ of n jobs that have to be scheduled on a single variable-speed processor. Each job J_i is specified by a release time r_i, a deadline d_i and a processing volume v_i, $1 \leq i \leq n$. In a feasible schedule, J_i must be completely processed within the time interval $[r_i, d_i)$. Preemption of jobs is allowed, i.e., the execution of a job may be stopped and resumed later. Again, the goal is to find a feasible schedule for the given job instance $\sigma = J_1, \ldots, J_n$ minimizing energy consumption.

In the considered multiprocessor setting, each of the m given processors can individually run at variable speed s; the associated power function is $P(s)$. We consider the specific family of functions $P(s) = s^\alpha$, where $\alpha > 1$, as well as general convex non-decreasing functions $P(s)$. *Job migration* is allowed, i.e. whenever a job is preempted it may be moved to a different processor. Hence, over time, a job may be executed on various processors as long as the respective processing intervals do not overlap. Executing a job simultaneously on two or more processors is not allowed. The goal is to construct a feasible schedule minimizing the total energy consumption incurred by all the processors.

Both the offline and the online scenarios are of interest. In the offline setting, all jobs and their characteristics are known in advance. We wish to construct optimal schedules minimizing energy consumption. In the online setting, jobs arrive over time. Whenever a new job J_i arrives at time r_i, its deadline d_i and processing volume v_i are known. However, future jobs J_k, with $k > i$, are unknown. We use competitive analysis to evaluate the performance of online strategies [50]. Recall that an online algorithm A is called c-competitive if, for any job sequence σ, the energy consumption of A is at most c times the consumption of an optimal offline schedule.

Previous work

We again focus our review on deadline-based dynamic speed scaling, as it was introduced by Yao et al. [55]. In the introduction of the previous chapter, we have already reviewed the work addressing single-processor environments in the offline setting. Here we will focus on previous results in (1) single-processor environments in the online setting, and (2) multi-processor environments. Recall, that most of the previous work addresses single-processor environments and power functions $P(s) = s^\alpha$, where $\alpha > 1$.

Again, in [55] Yao et al. first studied the offline problem and presented a polynomial time algorithm for computing optimal schedules. In the same paper, Yao et al. also presented two elegant online algorithms called *Optimal Available* and *Average Rate*. They proved that *Average Rate* achieves a competitive ratio of $(2\alpha)^\alpha/2$. The analysis is essentially tight as Bansal et al. [15] showed a nearly matching lower bound of $((2-\delta)\alpha)^\alpha/2$, where δ goes to zero as α tends to infinity. Bansal et al. [19] analyzed *Optimal Available* and, using a clever potential function, proved a competitiveness of exactly α^α. They also proposed a new strategy that attains a competitive ratio of $2(\frac{\alpha}{\alpha-1})e^\alpha$. As for lower bounds, Bansal et al. [18] showed that the competitiveness of any deterministic strategy is at least $e^{\alpha-1}/\alpha$. Furthermore, no online algorithm has been analyzed or designed for general convex power functions.

The framework by Yao et al. assumes that there is no upper bound on the allowed processor speed. Articles [16,26,46] study settings in which the processor has a maximum speed or only a finite set of discrete speed levels.

The only previous work addressing deadline-based scheduling in multi-processor systems is [6, 24, 34, 43]. Bingham and Greenstreet [24] show that, if job migration is allowed, the offline problem can be solved in polynomial time using linear programming. The result holds for general convex non-decreasing power functions. Lam et al. [43] study a setting with two speed-bounded processors.

They show online algorithms that are constant competitive w.r.t. energy minimization and throughput maximization. Papers [6, 34] assume that job migration is *not* allowed. In this case the offline problem is NP-hard, even if all jobs have the same processing volume [6]. A randomized B_α-approximation algorithm and a randomized $2(\frac{\alpha}{\alpha-1})e^\alpha B_\alpha$-competitive online algorithm are given in [34]. Here B_α is the α-th Bell number. All the latter results were developed for the family of power functions $P(s) = s^\alpha$.

Our contribution

In this chapter we investigate dynamic speed scaling in general multi-processor environments, assuming that job migration is allowed. Using migration, scheduling algorithms can take advantage of the parallelism given by a multi-processor system in an effective way. We present a comprehensive study addressing both the offline and the online scenario.

First in Section 3.1 we study the offline problem and develop an efficient polynomial time algorithm for computing optimal schedules. The algorithm works for general convex non-decreasing power functions $P(s)$. As mentioned above, Bingham and Greenstreet [24] showed that the offline problem can be solved using linear programming. However, the authors mention that the complexity of their algorithm is too high for most practical applications. Instead in this chapter we develop a strongly combinatorial algorithm that relies on repeated maximum flow computations.

Our algorithm is different from the single-processor strategy by Yao et al. [55]. In a series of phases, the algorithm partitions the jobs $J_1, \ldots, J_n$ into job sets $\mathcal{J}_1, \ldots, \mathcal{J}_p$ such that all jobs $J_k \in \mathcal{J}_i$ are processed at the same uniform speed s_i, $1 \leq i \leq p$. Each such job set is computed using maximum flow calculations. In order to construct a flow network, we have to identify various properties of a specific class of optimal schedules. A key property is that, knowing $\mathcal{J}_1, \ldots, \mathcal{J}_{i-1}$, one can exactly determine the number of processors to be allocated to $\mathcal{J}_i$. At the beginning of the phase computing $\mathcal{J}_i$, the algorithm conjectures that the set $\mathcal{J} = \{J_1, \ldots, J_n\} \setminus (\mathcal{J}_1, \ldots, \mathcal{J}_{i-1})$ of all remaining jobs forms the next set $\mathcal{J}_i$. If this turns out not to be the case, the algorithm repeatedly removes jobs $J_k \in \mathcal{J}$ that do not belong to $\mathcal{J}_i$. The crucial step in the correctness proof is to show that each of these job removals is indeed correct so that, when the process terminates, the true set $\mathcal{J}_i$ is computed.

In Section 3.2 we study the online problem and, as in the previous literature, focus on power functions $P(s) = s^\alpha$, where $\alpha > 1$. We adapt the two popular strategies *Optimal Available* and *Average Rate* to multi-processor environments. Algorithm *Optimal Available*, whenever a new job arrives, computes an optimal schedule for the remaining workload. This can be done using our offline algorithm.

We prove that *Optimal Available* is α^α-competitive, as in the single-processor setting. While the adaption of the algorithm is immediate, its competitive analysis becomes considerably more involved. We can extend the potential function analysis by Bansal et al. [19] but have to define a refined potential and prove several properties that specify how an optimal schedule changes in response to the arrival of a new job. As for the second algorithm *Average Rate*, we present a strategy that distributes load among the processors so that the densities $\delta_i = v_i/(d_i - r_i)$ of the active jobs are almost optimally balanced. We prove that *Average Rate* achieves a competitiveness of $(2\alpha)^\alpha/2 + 1$. Hence, compared to competitive ratio in the single-processor setting, the factor only increases by the additive constant of 1.

3.1 A combinatorial offline algorithm

We develop a strongly combinatorial algorithm for constructing optimal offline schedules in polynomial time. Let $\sigma = J_1, \ldots, J_n$ be any job sequence and $P(s)$ be an arbitrary convex non-decreasing power function. Lemma 13 below implies that there exist optimal schedules that use at most n different speeds, say speeds $s_1 > s_2 > \ldots > s_p$ where $p \leq n$. Our algorithm constructs such an optimal schedule in p phases, starting from an initially empty schedule S_0. Let S_{i-1} be the schedule obtained at the end of phase $i-1$, $1 \leq i \leq p$. In phase i the algorithm identifies the set $\mathcal{J}_i$ of jobs that are processed at speed s_i, $1 \leq i \leq p$. Schedule S_{i-1} is then extended by the jobs of $\mathcal{J}_i$ to form a new schedule S_i. The job set $\mathcal{J}_i$ and the extension of the schedule are determined using repeated maximum flow computations. Finally S_p is an optimal feasible schedule.

We present three lemmas that we will also use when analyzing an extension of *Optimal Available* in Section 3.2. In that section we will consider power functions $P(s) = s^\alpha$, where $\alpha > 1$.

In Chapter 2, and for the single processor case, we assumed without loss of generality that every job is processed at its own constant speed. This assumption can also be made in the multiprocessor setting.

Lemma 13. *Any optimal schedule can be modified such that every job J_i, $1 \leq i \leq n$, is processed at a constant non-varying speed. During the modification the schedule remains feasible and the energy consumption does not increase.*

Proof. Consider any optimal schedule. While there exists a job J_i, $1 \leq i \leq n$, that is processed at non-constant speed, let s_i be the average speed at which the job is executed. In the intervals in which J_i is processed modify the schedule so that J_i is executed at constant speed s_i on the same processor. The schedule remains feasible, since the total processing volume v_i of J_i is still completed in these intervals, and no job is being processed on two different processors simultaneously.

By the convexity of the power function $P(s)$ the modification does not increase the energy consumption of the schedule. $\square$

By the above lemma we restrict ourselves to optimal schedules that process each job at a constant speed. Let $\mathcal{S}_{OPT}$ be the set of such optimal schedules. In any schedule $S_{OPT} \in \mathcal{S}_{OPT}$ the n jobs $J_1, \ldots, J_n$ can be partitioned into sets $\mathcal{J}_1, \ldots, \mathcal{J}_p$ such that all the jobs of $\mathcal{J}_i$ are processed at the same speed s_i, $1 \leq i \leq p$. Each job belongs to exactly one of these sets.

In the set $\mathcal{S}_{OPT}$ we identify a subset $\mathcal{S}'_{OPT}$ of schedules having some favorable properties. If the power function is $P(s) = s^\alpha$, with $\alpha > 1$, then let $\mathcal{S}'_{OPT} = \mathcal{S}_{OPT}$. Otherwise, if $P(s)$ is a different power function, fix any $\alpha > 1$ and let $P_\alpha(s) = s^\alpha$ be an auxiliary power function. Let $\mathcal{S}'_{OPT} \subseteq \mathcal{S}_{OPT}$ be the subset of the schedules that, among schedules of $\mathcal{S}_{OPT}$, incur the smallest energy consumption if $P(s)$ is replaced by $P_\alpha(s)$. That is, among the schedules in $\mathcal{S}_{OPT}$, we consider those that would yield the smallest energy consumption if energy was accounted according to $P_\alpha(s)$. Since set $\mathcal{S}_{OPT}$ is non-empty, subset $\mathcal{S}'_{OPT}$ is also non-empty. Such an auxiliary power function was also used in [39] to break ties among optimal schedules.

The next lemma ensures that any schedule of $\mathcal{S}'_{OPT}$ can be modified such that the processor speeds change only at the release times and deadlines of jobs. Let $\mathcal{I} = \{r_i, d_i \mid 1 \leq i \leq n\}$ be the set of all release times and deadlines. We consider the elements of $\mathcal{I}$ in sorted order $\tau_1 < \ldots < \tau_{|\mathcal{I}|}$, where $|\mathcal{I}| \leq 2n$. The time horizon in which jobs can be scheduled is $[\tau_1, \tau_{|\mathcal{I}|})$. We partition this time horizon along the job release times and deadlines into intervals $I_j = [\tau_j, \tau_{j+1})$, $1 \leq j < |\mathcal{I}|$. Let $|I_j| = \tau_{j+1} - \tau_j$ be the length of I_j.

Lemma 14. *Given any optimal schedule $S_{OPT} \in \mathcal{S}'_{OPT}$, in each interval I_j we can rearrange the schedule such that every processor uses a constant, non-varying speed in I_j, $1 \leq j < |\mathcal{I}|$. During the modification the schedule remains feasible and the energy consumption with respect to $P(s)$ and $P_\alpha(s)$ does not increase.*

Proof. Consider an arbitrary optimal schedule $S_{OPT} \in \mathcal{S}'_{OPT}$ and let I_j be any interval, $1 \leq j < |\mathcal{I}|$. We show how to modify the schedule in I_j, without changing the energy consumption, so that the desired property holds for I_j. Modifying all the intervals in that way, we obtain the lemma.

For a given I_j, let $S_{OPT}(I_j)$ be the schedule of S_{OPT} restricted to I_j. Moreover, let $s_{j_1} > s_{j_2} > \ldots > s_{j_l}$ be the different speeds employed in I_j. Assume that s_{j_k} is used for t_k time units in I_j, considering all the m processors, and let $\mathcal{J}_{j_k}(I_j)$ be the set of jobs processed at speed s_{j_k} in I_j, $1 \leq k \leq l$. Since jobs are processed at constant speed the sets $\mathcal{J}_{j_k}(I_j)$, $1 \leq k \leq l$, are disjoint. For a job $J_i \in \mathcal{J}_{j_k}(I_j)$, let $t_{i,j}$ be the total time for which J_i is scheduled in I_j. We have $t_{i,j} \leq |I_j|$ since otherwise $S_{OPT}(I_j)$ and hence S_{OPT} would not be feasible. Moreover, $\sum_{J_i \in \mathcal{J}_{j_k}(I_j)} t_{i,j} = t_k$.

We next construct a *working schedule* W of length $\sum_{k=1}^{l} t_k$ for the jobs of $\mathcal{J}_{j_1}(I_j), \ldots, \mathcal{J}_{j_l}(I_j)$ and then distribute it among the processors. More specifically, for any k with $1 \leq k \leq l$, we construct a schedule W_k of length t_k in which the jobs of $\mathcal{J}_{j_k}(I_j)$, using a speed of s_{j_k}, are scheduled as follows: First the jobs J_i with $t_{i,j} = |I_j|$ are processed, followed by the jobs J_i with $t_{i,j} < |I_j|$. Each J_i is assigned $t_{i,j}$ consecutive time units in W_k. We concatenate the W_k to form the working schedule $W = W_1 \circ \ldots \circ W_l$ of length $\sum_{k=1}^{l} t_k \leq m|I_j|$. Next we distribute W among the m processors by assigning time window $[(\mu-1)|I_j|, \mu|I_j|)$ of W to processor μ, for $1 \leq \mu \leq \lceil \sum_{k=1}^{l} t_k / |I_j| \rceil$. Each window is scheduled from left to right on the corresponding processor, starting at the beginning of I_j. As we shall prove below, the total length of W is an integer multiple of $|I_j|$ so that exactly $\lceil \sum_{k=1}^{l} t_k / |I_j| \rceil$ processors are filled, without any idle period at the end. Let $S'_{OPT}(I_j)$ be the new schedule for I_j. Furthermore, let S'_{OPT} be the schedule obtained from S_{OPT} when replacing $S_{OPT}(I_j)$ by $S'_{OPT}(I_j)$. Schedule S_{OPT} is feasible because, for each job, the total execution time and employed speed have not changed. If in $S'_{OPT}(I_j)$ the execution of a job $J_i \in \mathcal{J}_{j_k}(I_j)$ is split among two processors μ and $\mu + 1$, then J_i is processed at the end of I_j on processor μ and at the beginning of I_j on processor $\mu + 1$. The two execution intervals do not overlap because $t_{i,j} \leq |I_j|$. Furthermore, the energy consumption of S'_{OPT} is the same as that of S_{OPT} because the processing intervals of the jobs have only been rearranged. This holds true with respect to both $P(s)$ and $P_\alpha(s)$.

To establish the statement of the lemma for I_j we show that each t_k is an integer multiple of $|I_j|$, i.e. $t_k = m_k|I_j|$ for some positive integer m_k, $1 \leq k \leq l$. This implies that the new schedule never uses two or more speeds on any processor. So suppose that some t_k is not in integer multiple of $|I_j|$. Consider the smallest index k with this property. Then in $S'_{OPT}(I_j)$, on the corresponding processor, the processing of W_k ends at some time t strictly before the end of I_j, i.e. $t < \tau_{j+1}$. Furthermore, the job J_i handled last in W_k is processed for less than $|I_j|$ time units since otherwise all jobs in W_k would be executed for $|I_j|$ time units, contradicting the choice of k. So let $t_{i,j} = |I_j| - \delta_i$, for some $\delta_i > 0$. Set $\delta = \min\{t - \tau_j, \tau_{j+1} - t, \delta_i, t_{i,j}\}$. We now modify $S'_{OPT}(I_j)$ on the processor handling the end of J_i within I_j. During the last δ time units before t we decrease the speed by ϵ, and during the first δ time units after t we increase the speed by ϵ, where $\epsilon = (s_{j_k} - s_{j_{k+1}})/2$. If $k = l$, we set $s_{j_{k+1}} = 0$, assuming that the processing of W_l is followed by an idle period. Since $\delta \leq \tau_{j+1} - t$ and $\delta \leq t - \tau_j$, the modifications only affect the processor under consideration. Furthermore, the total work finished in $[t - \delta, t + \delta]$ remains unchanged. As the speed was lowered in $[t - \delta, t)$, the first time units after t are used to finish the work on J_i. This can be accomplished before time $t + \delta \leq t + \delta_i$ so that J_i is not processed for more than $|I_j|$ time units and the resulting schedule is feasible.

Let S^m_{OPT} be the modified schedule obtained. Since $P(s)$ is convex, S^m_{OPT} does not consume more energy than S'_{OPT} with respect to $P(s)$. Hence S^m_{OPT} is an optimal schedule. However, since $\delta s^\alpha_{j_k} + \delta s^\alpha_{j_{k+1}} > 2\delta((s_{j_k} + s_{j_{k+1}})/2)^\alpha$ schedule S^m_{OPT} consumes strictly less energy than S'_{OPT} with respect to $P_\alpha(s)$. Finally, modify S^m_{OPT} so that each job is processed at constant speed. This can be done by processing job J_i and the job possibly scheduled after J_i in I_j using their respective average speeds. Since $P(s)$ and $P_\alpha(s)$ are convex, the energy consumption does not increase. The resulting schedule belongs to set $\mathcal{S}_{OPT}$ and consumes strictly less energy than schedules S'_{OPT} and S_{OPT} with respect to $P_\alpha(s)$. We obtain a contradiction to the fact that $S_{OPT} \in \mathcal{S}'_{OPT}$. □

In the remainder of this section we consider optimal schedules $S_{OPT} \in \mathcal{S}'_{OPT}$ satisfying the property of Lemma 14. Since set $\mathcal{S}'_{OPT}$ is non-empty, these schedules always exist. As described above we will construct such a schedule S_{OPT} in phases, where phase i identifies the set $\mathcal{J}_i$ of jobs processed at speed s_i, $1 \leq i \leq p$. By Lemma 14, in each interval I_j, $1 \leq j < |\mathcal{I}|$, the sets $\mathcal{J}_1, \ldots, \mathcal{J}_p$ occupy different processors, i.e. there is no processor handling jobs of two different sets. Hence, in phase i, when determining $\mathcal{J}_i$ and extending the previous schedule S_{i-1}, we only need to know the number of processors occupied by $\mathcal{J}_1, \ldots, \mathcal{J}_{i-1}$ in any interval I_j, $1 \leq j < |\mathcal{I}|$. The exact assignment of the corresponding jobs to the reserved processors is irrelevant though, as $\mathcal{J}_i$ does not share processors with the previous sets. In determining $\mathcal{J}_i$ a crucial question is, how many processors to allocate to the jobs of $\mathcal{J}_i$ in any interval I_j, $1 \leq j < |\mathcal{I}|$. Here the following lemma is essential.

A job J_k, $1 \leq k \leq n$, is called *active* in I_j if $I_j \subseteq [r_k, d_k)$, i.e. the time period in which J_k can be scheduled includes I_j. Given schedule S_{OPT}, let n_{ij} be the number of jobs of $\mathcal{J}_i$ that are active in I_j. Furthermore, let m_{ij} denote the number of processors occupied by $\mathcal{J}_i$ in I_j.

Lemma 15. *Let $S_{OPT} \in \mathcal{S}'_{OPT}$ be any schedule satisfying the property of Lemma 14. Then the jobs of $\mathcal{J}_i$ occupy $m_{ij} = \min\{n_{ij}, m - \sum_{l=1}^{i-1} m_{lj}\}$ processors in I_j, where $1 \leq i \leq p$ and $1 \leq j < |\mathcal{I}|$.*

Proof. Consider a set $\mathcal{J}_i$ and an interval I_j. Obviously, the n_{ij} jobs of $\mathcal{J}_i$ that are active in I_j cannot occupy more than n_{ij} processors using a positive speed s_i throughout I_j. Furthermore, the jobs can only occupy the $m - \sum_{l=1}^{i-1} m_{lj}$ processors not taken by $\mathcal{J}_1, \ldots, \mathcal{J}_{i-1}$. We show that in fact $m_{ij} = \min\{n_{ij}, m - \sum_{l=1}^{i-1} m_{lj}\}$ processors are used by $\mathcal{J}_i$

So suppose that m'_{ij} processors, where $m'_{ij} < m_{ij}$, are used. Then, since $m'_{ij} < m - \sum_{l=1}^{i-1} m_{lj}$, there exists at least one processor P running at speed s

with $s < s_i$ in I_j. Consider the schedule at the beginning of I_j. Choose a δ, $\delta > 0$, such that the m'_{ij} processors handling $\mathcal{J}_i$ and processor P do not preempt jobs in $[\tau_j, \tau_j + \delta)$, i.e. they each handle at most one job in that time window. In fact P might not handle a job if $s = 0$. As $m'_{ij} < n_{ij}$, there must exist one job $J_k \in \mathcal{J}_i$ that is active in I_j but not scheduled within $[\tau_j, \tau_j + \delta)$ on any of the m'_{ij} processors using speed s_i. This job is not scheduled on any other processor within $[\tau_j, \tau_j + \delta)$ either, because the other processors run at speeds higher or lower than s_i. Thus J_k is not executed on any processor within $[\tau_j, \tau_j + \delta)$. In the entire schedule, consider a processor and an associated time window W of length $\delta' \leq \delta$ handling J_k. We now modify the schedule by reducing the speed in W by ϵ, where $\epsilon = (s_i - s)/2$. At the same time we increase the speed on processor P in $[\tau_j, \tau_j + \delta')$ by ϵ and use the extra processing capacity to complete the missing portion of J_k not finished in W. Let S^m_{OPT} be the modified schedule. Since $P(s)$ is convex, S^m_{OPT} does not consume more energy than S_{OPT} with respect to $P(s)$. With respect to $P_\alpha(s)$, schedule S^m_{OPT} consumes less energy than S_{OPT} because $\delta' s_i^\alpha + \delta' s^\alpha > 2\delta'((s_i + s)/2)^\alpha$. Finally, modify S^m_{OPT} so that each job is processed at a constant speed. We obtain a schedule that belongs to set $\mathcal{S}_{OPT}$ and consumes strictly less energy than schedule S_{OPT} with respect to $P_\alpha(s)$. This contradicts the fact that $S_{OPT} \in \mathcal{S}'_{OPT}$. $\square$

Lemma 15 has an interesting implication: Suppose that job sets $\mathcal{J}_1, \ldots, \mathcal{J}_{i-1}$ along with the number of occupied processors $m_{1j}, \ldots, m_{i-1,j}$, for $1 \leq j < |\mathcal{I}|$, have been determined. Furthermore, suppose that the set $\mathcal{J}_i$ is known. Then, using Lemma 15, we can immediately determine the number m_{ij} of processors used by $\mathcal{J}_i$ in I_j, $1 \leq j < |\mathcal{I}|$. Moreover, we can compute the speed s_i by observing that, since $P(s)$ is non-decreasing, s_i can be set to the minimum average speed necessary to complete the jobs of $\mathcal{J}_i$ in the reserved processing intervals. More precisely, let $V_i = \sum_{J_k \in \mathcal{J}_i} v_k$ be the total processing volume of $\mathcal{J}_i$ and $P_i = \sum_{1 \leq j < |\mathcal{I}|} m_{ij}|I_j|$ be the total length of the reserved processing intervals. Then $s_i = V_i / P_i$. We will make use of this fact when identifying $\mathcal{J}_i$.

Description of the algorithm: We show how each phase i, determining job set $\mathcal{J}_i$, works. Assume again that $\mathcal{J}_1, \ldots, \mathcal{J}_{i-1}$ along with processor numbers $m_{1j}, \ldots, m_{i-1,j}$, for $1 \leq j < |\mathcal{I}|$, are given. In phase i the algorithm operates in a series of rounds, maintaining a job set $\mathcal{J}$ that represents the current estimate for the true $\mathcal{J}_i$. At any time the invariant $\mathcal{J}_i \subseteq \mathcal{J}$ holds. Initially, prior to the first round, the algorithm sets $\mathcal{J} := \{J_1, \ldots, J_n\} \setminus (\mathcal{J}_1 \cup \ldots \cup \mathcal{J}_{i-1})$, which is the set of all remaining jobs, and the invariant is obviously satisfied. In each round the algorithm checks if the current $\mathcal{J}$ is the desired $\mathcal{J}_i$. This can be verified using a maximum flow computation. If $\mathcal{J}$ turns out to be $\mathcal{J}_i$, then phase i stops. Otherwise the algorithm determines a job $J_k \in \mathcal{J} \setminus \mathcal{J}_i$, removes that job from $\mathcal{J}$ and starts the next round in which the updated set $\mathcal{J}$ is checked.

In the following we describe the maximum flow computation invoked for a given $\mathcal{J}$. First the algorithm determines the number of processors to be allocated to $\mathcal{J}$ in each interval. For any I_j, $1 \le j < |\mathcal{I}|$, let n_j be the number of jobs in $\mathcal{J}$ that are active in $|I_j|$. According to Lemma 15 the algorithm reserves $m_j = \min\{n_j, m - \sum_{l=1}^{i-1} m_{lj}\}$ processors in I_j. These numbers form a vector $\vec{m} = (m_j)_{1\le j<|\mathcal{I}|}$. The speed is set to $s = V/P$, where $V = \sum_{J_k\in\mathcal{J}} v_k$ is the total processing volume and $P = \sum_{1\le j<|\mathcal{I}|} m_j|I_j|$ is the reserved processing time.

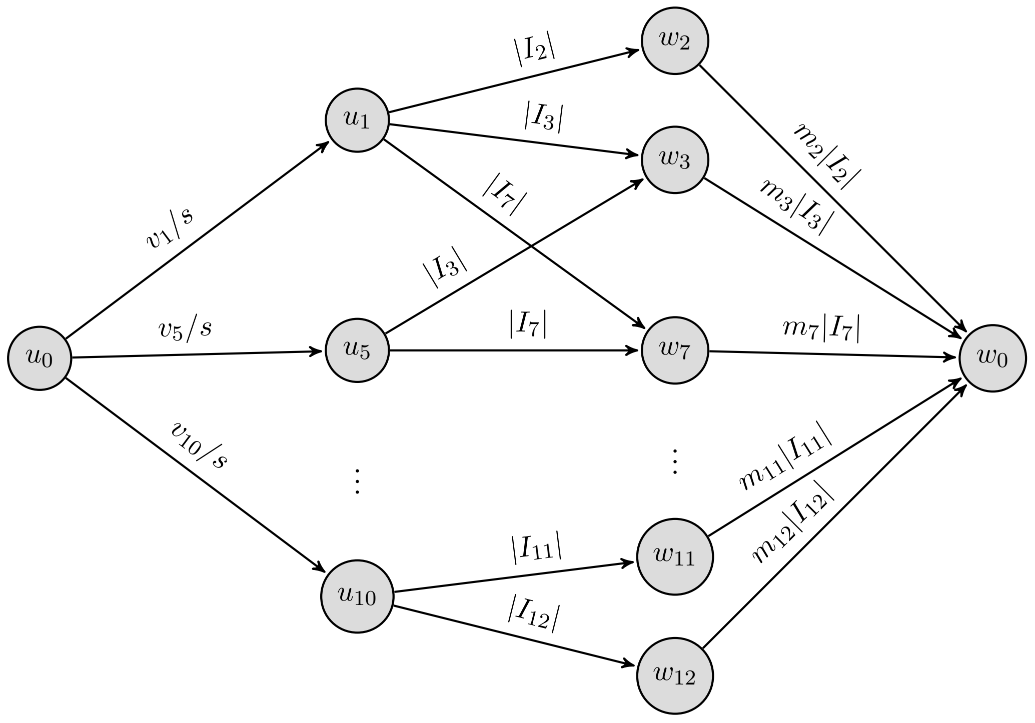


Figure 3.1: The basic structure of $G(\mathcal{J},\vec{m},s)$, assuming that set $\mathcal{J} = \{J_1, J_5, \dots, J_{10}\}$ can be scheduled in intervals $I_2,\ I_3,\ I_7,\ \dots,\ I_{11},\ I_{12}$.

Next we define the graph $G(\mathcal{J},\vec{m},s)$ of the maximum flow computation. For each job $J_k \in \mathcal{J}$ we introduce a vertex u_k, and for each interval I_j with $m_j > 0$ we introduce a vertex w_j. For any u_k and w_j such that J_k is active in I_j we add a directed edge (u_k, w_j) of capacity $|I_j|$. Hence each u_k is connected to exactly those w_j such that J_k can be scheduled in I_j. The edge capacity of (u_k, w_j) is equal to $|I_j|$, i.e. the maximum time for which a job can be processed in I_j. Furthermore, we introduce a source vertex u_0 that is connected to every u_k on a directed edge (u_0, u_k). The edge capacity is set to v_k/s, which is the total processing time needed for J_k using speed s. Finally we introduce a sink w_0 and connect each w_j to w_0 via a directed edge (w_j, w_0) of capacity $m_j|I_j|$, which is the total processing time available on the reserved processors in I_j. The global structure of $G(\mathcal{J}, \vec{m}, s)$ is depicted in Figure 3.1. The value of a maximum flow in $G(\mathcal{J}, \vec{m}, s)$ is upper bounded by $F_G = \sum_{J_k \in \mathcal{J}} v_k/s$ because this is the total capacity of the edges leaving the source u_0. This is equal to the total capacity of the edges leading into the sink w_0 because that value is

$$\sum_{1 \le j < |\mathcal{I}|} m_j |I_j| = P = V/s = \sum_{J_k \in \mathcal{J}} v_k/s = F_G.$$

We argue that job set $\mathcal{J}$ using a continuous speed of s can be feasibly scheduled on the reserved processors represented by $\vec{m}$ if and only if there exists a maximum flow of value F_G in $G(\mathcal{J}, \vec{m}, s)$. First suppose that there exists a feasible schedule using speed s. Then each $J_k \in \mathcal{J}$ is processed for v_k/s time units and we send v_k/s units of flow along edge (u_0, u_k). The v_k/s processing units of J_k are scheduled in intervals I_j in which the job is active. If J_k is processed for t_{kj} time units in I_j, then we send t_{kj} units of flow along edge (u_k, w_j). The edge capacity is observed because $t_{kj} \le |I_j|$. The total amount of flow entering and hence leaving any w_j is equal to the total processing time on the m_j reserved processor in I_j. This value is equal to $m_j |I_j|$ because, by the choice of s, all reserved processors are busy throughout the execution intervals. Hence the amount of flow sent along (w_j, w_0) is equal to the edge capacity.

On the other hand, assume that a flow of value F_G is given. If t_{kj} units of flow are routed along (u_k, w_j), we process J_k for t_{kj} time units in I_j. This is possible because J_k is active in I_j and $t_{kj} \le |I_j|$. The capacity constraints on the edge (w_j, w_0) ensure that not more than $m_j I_j$ time units are assigned to I_j. Moreover, since the flow value is F_G, each edge (u_0, u_k) is saturated and hence J_k is fully processed for v_k/s time units at speed s. Within each I_j we can easily construct a feasible schedule by concatenating the processing intervals of length t_{kj} associated with the jobs J_k active in I_j. The resulting sequential schedule, say S, of length $\sum_{J_k \in \mathcal{J}} t_{kj} = m_j |I_j|$ is split among the m_j reserved processors by assigning time range $[(\mu - 1)|I_j|, \mu |I_j|)$ of S to the μ-th reserved processor, $1 \le \mu \le m_j$.

We proceed to describe how the algorithm determines $\mathcal{J}_i$. In each round, for the current set $\mathcal{J}$, the algorithm invokes a maximum flow computation in the graph $G(\mathcal{J}, \vec{m}, s)$ defined in the previous paragraphs. If the flow value is equal to F_G, then the algorithm stops and sets $\mathcal{J}_i := \mathcal{J}$. We will prove below that in this case the current $\mathcal{J}$ is indeed equal to $\mathcal{J}_i$. If the flow value is smaller than F_G, then there must exist an edge (w_j, w_0) into the sink carrying less than $m_j |I_j|$ units of flow. Hence there must exist an edge (u_k, w_j) carrying less than $|I_j|$ units of flow because at least m_j jobs of $\mathcal{J}$ are active in I_j. The algorithm removes the corresponding job J_k, i.e. $\mathcal{J} := \mathcal{J} \setminus \{J_k\}$. We will also prove below that this removal is correct, i.e. $\mathcal{J}_i$ does not contain J_k. For the new set $\mathcal{J}$ the algorithm starts the next round, invokes a maximum flow computation in the updated graph $G(\mathcal{J}, \vec{m}, s)$, where $\vec{m}$ and s are updated too. The sequence of rounds ends when finally a flow of value F_G in the current graph $G(\mathcal{J}, \vec{m}, s)$ is found. A formal description of the entire algorithm is given in Figure 3.2. The pseudo-code uses a boolean variable set_found that keeps track whether or not the desired $\mathcal{J}_i$ has been correctly determined.

```
Algorithm Optimal Schedule
 1. $\mathcal{J} := \{J_1, \ldots, J_n\}$;  $i := 0$;  $S_0$ := empty schedule;
 2. while $\mathcal{J} \neq \emptyset$ do
 3.     $i := i + 1$; set_found := false;
 4.     while ¬set_found do
 5.         $n_j$ := number of jobs of $\mathcal{J}$ that are active in $I_j$, for $1 \leq j < |\mathcal{I}|$;
 6.         $m_j := \min\{n_j, \sum_{l=1}^{i-1} m_{lj}\}$; $\vec{m} := (m_j)_{1 \leq j < |\mathcal{I}|}$;
 7.         $V := \sum_{J_k \in \mathcal{J}} v_k$;  $P := \sum_{1 \leq j < |\mathcal{I}|} m_j |I_j|$; $s := V/P$;
 8.         Compute the value $F$ of a maximum flow in $G(\mathcal{J}, \vec{m}, s)$;
 9.         if $F = V/s$ then set_found := true;
10.         else Determine edge $(w_j, w_0)$ carrying less than $m_j|I_j|$ units of
                 flow and edge $(u_k, w_j)$ carrying less than $|I_j|$ units of flow;
                 Set $\mathcal{J} := \mathcal{J} \setminus \{J_k\}$;
11.     $\mathcal{J}_i := \mathcal{J}$;  $s_i := s$;  $m_{ij} := m_j$ for $1 \leq j < |\mathcal{I}|$;
12.     Extend $S_{i-1}$ by a feasible schedule for $\mathcal{J}_i$ using speed $s_i$ and $m_{ij}$
          processors in $I_j$;  Let $S_i$ be the resulting schedule;
13.     $\mathcal{J} := \{J_1, \ldots, J_n\} \setminus (\mathcal{J}_1 \cup \ldots \cup \mathcal{J}_i)$;
```

Figure 3.2: The entire offline algorithm for computing an optimal schedule.

It remains to prove correctness of our algorithm. We prove that in any phase i, $\mathcal{J}_i$ is correctly determined. Lemma 15 immediately implies that the number m_{ij} of reserved processors, for $1 \leq j < |\mathcal{I}|$, as specified in lines 6 and 11 of the algorithm is correct. A schedule for $\mathcal{J}_i$ on the reserved processors can be derived easily from the maximum flow computed in line 8. We described the construction of this schedule above when arguing that a flow of maximum value F_G in $G(\mathcal{J}, \vec{m}, s)$ implies a feasible schedule using speed s on the reserved processors $\vec{m}$. It remains to show that sets $\mathcal{J}_1, \ldots, \mathcal{J}_p$ are correct. The proof is by induction on i. Consider an i, $1 \leq i \leq p$, and suppose that $\mathcal{J}_1, \ldots, \mathcal{J}_{i-1}$ have been computed correctly. The inductive step and hence correctness of $\mathcal{J}_i$ follow from Lemmas 16 and 17.

Lemma 16. *In phase i, the set $\mathcal{J}$ maintained by the algorithm always satisfies the invariant $\mathcal{J}_i \subseteq \mathcal{J}$.*

Proof. At the beginning of the phase $\mathcal{J} = \{J_1, \dots, J_n\} \setminus (\mathcal{J}_1, \dots, \mathcal{J}_{i-1})$ is the set of all remaining jobs and the invariant is satisfied. Consider a round, represented by lines 5 to 10 of the algorithm, where in the beginning $\mathcal{J}_i \subseteq \mathcal{J}$ holds and a job J_{k_0} is removed in line 10 at the end of the round. We prove that J_{k_0} does not belong to $\mathcal{J}_i$.

Consider the flow computed in line 8 of the round. We call a vertex w_j *saturated* if the amount of flow routed along the outgoing edge (w_j, w_0) is equal to the capacity $m_j|I_j|$. Otherwise w_j is *unsaturated*. Since the computed flow has value $F < F_G = V/s = P = \sum_{1 \le j < |\mathcal{I}|} m_j|I_j|$, there exists at least one unsaturated vertex. We now modify the flow so as to increase the number of unsaturated vertices. For an edge e in the graph, let $f(e)$ be the amount of flow routed along it. The modification is as follows: While there exists a vertex u_k with flow $f(u_k, w_j) < |I_j|$ into an unsaturated vertex w_j and positive flow $f(u_k, w_{j'}) > 0$ into a saturated vertex $w_{j'}$, change the flow as follows. Along (u_k, w_j) and (w_j, w_0) we increase the flow by ϵ and along $(u_k, w_{j'})$ and $(w_{j'}, w_0)$ we reduce the flow by ϵ, where $\epsilon = \frac{1}{2}\min\{f(u_k, w_{j'}), |I_j| - f(u_k, w_j), m_j|I_j| - f(w_j, w_0)\}$. At u_k, w_j and $w_{j'}$ the flow conservation law is observed. By the choice of ϵ, the new flow along $(u_k, w_{j'})$ and $(w_{j'}, w_0)$ is positive. Also, by the choice of ϵ, the capacity constraints on (u_k, w_j) and (w_j, w_0) are not violated. Hence the new flow is feasible. Moreover, the total value F of the flow does not change. Vertex $w_{j'}$ is a new unsaturated vertex while w_j continues to be unsaturated because $\epsilon < m_j|I_j| - f(w_j, w_0)$ and hence the new flow along (w_j, w_0) is strictly smaller than the capacity. Also, since $\epsilon < |I_j| - f(u_k, w_j)$, the new flow along (u_k, w_j) remains strictly below $|I_j|$. The modifications stop when the required conditions are not satisfied anymore, which happens after less than $|\mathcal{I}|$ steps.

Given the modified flow, let $U = \{j \mid w_j \text{ is unsaturated}\}$ be the indices of the unsaturated vertices. Intuitively, these vertices correspond to the intervals in which less than $m_j|I_j|$ units of processing time would be scheduled. Let $\mathcal{J}'$, $\mathcal{J}' \subseteq \mathcal{J}$, be the set of jobs J_k such that $f(u_0, u_k) = v_k/s$ and the outgoing edges leaving u_k that carry positive flow all lead into unsaturated vertices. We argue that the job J_{k_0} removed from $\mathcal{J}$ in line 10 of the algorithm belongs to $\mathcal{J}'$: Prior to the flow modifications, there was an unsaturated vertex w_{j_0} having an incoming edge (u_{k_0}, w_{j_0}) with $f(u_{k_0}, w_{j_0}) < |I_{j_0}|$. Throughout the flow modifications, w_{j_0} remains an unsaturated vertex, and any flow update on (u_{k_0}, w_{j_0}) ensures that the amount of flow is strictly below $|I_{j_0}|$. Hence, if u_{k_0} had an outgoing edge with positive flow into a saturated vertex, the flow modifications would not have stopped. Moreover, if $f(u_0, u_{k_0}) < v_{k_0}/s$ held, we could increase the flow along edges (u_0, u_{k_0}), (u_{k_0}, w_{j_0}) and (w_{j_0}, w_0), thereby increasing the total value F of the flow.

Next let $\mathcal{J}''$, $\mathcal{J}'' \subseteq \mathcal{J} \setminus \mathcal{J}'$, be the set of jobs J_k such that u_k has at least one edge (u_k, w_j) into an unsaturated vertex w_j. Here we argue that for any such job, all the edges (u_k, w_j) into unsaturated vertices w_j carry a flow of exactly $|I_j|$ units: First observe that a job $J_k \in \mathcal{J}''$ does not belong to $\mathcal{J}'$ and hence (a) there is an edge (u_k, w_j) with positive flow into a saturated vertex w_j or (b) $f(u_0, u_k) < v_k/s$. In case (a), if there was an edge (u_k, w_j) carrying less than $|I_j|$ units of flow into a unsaturated vertex w_j, the flow modifications would not have stopped. In case (b), we could increase the flow along (u_0, u_k), (u_k, w_j) and (w_j, w_0), thereby raising the flow value F.

For any I_j, let $\mathcal{J}'(I_j)$, be the set of jobs of $\mathcal{J}'$ that are active in I_j. Set $\mathcal{J}''(I_j)$ is defined analogously. Given the modified flow, let t_{kj} be the amount of flow along (u_k, w_j). Note that for no job $J_k \in \mathcal{J} \setminus (\mathcal{J}' \cup \mathcal{J}'')$ there exists an edge (u_k, w_j) into a unsaturated vertex and hence J_k is not active in an I_j with $j \in U$. Hence, for any $j \in U$, since w_j is unsaturated, $\sum_{J_k \in \mathcal{J}'(I_j) \cup \mathcal{J}''(I_j)} t_{kj} < m_j|I_j|$. As $t_{kj} = |I_j|$, for $J_k \in \mathcal{J}''(I_j)$,

$$\sum_{J_k \in \mathcal{J}'(I_j)} t_{kj} < (m_j - |\mathcal{J}''(I_j)|)|I_j|. \tag{3.1}$$

We next prove that the job J_{k_0} removed at the end of the round in line 10 of the algorithm does not belong to $\mathcal{J}' \cap \mathcal{J}_i$ and hence does not belong to $\mathcal{J}_i$ because, as argued above, $J_{k_0} \in \mathcal{J}'$. So suppose that $J_{k_0} \in \mathcal{J}' \cap \mathcal{J}_i$. We show that in this case the total processing volume of $\mathcal{J}_i$ is not large enough to fill the reserved processing intervals using a continuous speed of s_i. We observe that s_i is at least as large as the speed s computed in the current round of the algorithm: If $s_i < s$, then in the optimal schedule, all jobs of $\mathcal{J}$ would be processed at a speed smaller than s. However, the total length of the available processing intervals for $\mathcal{J}$ is not more than $P = \sum_{1 \le j < |\mathcal{I}|} m_j|I_j|$ and total work to be completed is $V = \sum_{J_k \in \mathcal{J}} v_k$. Hence a speed of $s_i < s = V/P$ is not sufficient to finish the jobs in time.

Let m_{ij} be the number of processors used by $\mathcal{J}_i$ in interval I_j in an optimal solution. We prove that the jobs of $\mathcal{J}_i$ cannot fill all the reserved processors in intervals I_j with $j \in U$ using a speed of $s_i \ge s$. Recall that no job of $\mathcal{J} \setminus (\mathcal{J}' \cup \mathcal{J}'')$ is active in any interval I_j with $j \in U$. Only the jobs of $\mathcal{J}'(I_j) \cap \mathcal{J}_i$ and $\mathcal{J}''(I_j) \cap \mathcal{J}_i$ are active. The jobs of $\mathcal{J}'(I_j) \cap \mathcal{J}_i$ must fill at least $m_{ij} - |\mathcal{J}''(I_j) \cap \mathcal{J}_i|$ processors over a time window of length $|I_j|$ because each job of $\mathcal{J}''(I_j) \cap \mathcal{J}_i$ can fill at best one processor. Thus, over all intervals I_j with $j \in U$, the jobs of $\mathcal{J}' \cap \mathcal{J}_i$ have to provide a work of at least

$$\sum_{j \in U} (m_{ij} - |\mathcal{J}''(I_j) \cap \mathcal{J}_i|)|I_j| s_i. \tag{3.2}$$

We show that this is not the case. The total work of $\mathcal{J}' \cap \mathcal{J}_i$ is

$$\sum_{J_k \in \mathcal{J}' \cap \mathcal{J}_i} v_k = \sum_{J_k \in \mathcal{J}' \cap \mathcal{J}_i} f(u_0, u_k) s = \sum_{j \in U} \sum_{J_k \in \mathcal{J}' \cap \mathcal{J}_i} t_{kj} s, \tag{3.3}$$

where t_{kj} is again the amount of flow along (u_k, w_j) in the modified flow. The first and second equations hold because, for any $J_k \in \mathcal{J}'$, $f(u_0, u_k) = v_k/s$ and along edges (u_k, w_j) positive amounts of flow are routed only into unsaturated vertices w_j, $j \in U$. For any $j \in U$, we have

$$\sum_{J_k \in \mathcal{J}'(I_j) \cap \mathcal{J}_i} t_{kj} \leq |\mathcal{J}'(I_j) \cap \mathcal{J}_i||I_j|$$

because the edge capacity of (u_k, w_j) is $|I_j|$. For the unsaturated vertex w_{j_0} and the job J_{k_0}, determined in line 10 of the algorithm, the flow along (u_{k_0}, w_{j_0}) is strictly below $|I_{j_0}|$. Hence for $j_0 \in U$ the stronger relation

$$\sum_{J_k \in \mathcal{J}'(I_{j_0}) \cap \mathcal{J}_i} t_{kj_0} < |\mathcal{J}'(I_{j_0}) \cap \mathcal{J}_i||I_{j_0}|$$

holds because by assumption $J_{k_0} \in \mathcal{J}' \cap \mathcal{J}_i$. Taking into account (3.1), we obtain that expression (3.3) is

$$\sum_{J_k \in \mathcal{J}' \cap \mathcal{J}_i} v_k < \sum_{j \in U} \min\{|\mathcal{J}'(I_j) \cap \mathcal{J}_i|, m_j - |\mathcal{J}''(I_j)|\}|I_j|s.$$

We will show that for any $j \in U$,

$$\min\{|\mathcal{J}'(I_j) \cap \mathcal{J}_i|, m_j - |\mathcal{J}''(I_j)|\} \leq m_{ij} - |\mathcal{J}''(I_j) \cap \mathcal{J}_i|. \tag{3.4}$$

Since $s_i \geq s$ this implies that the total work of $\mathcal{J}' \cap \mathcal{J}_i$ is strictly smaller than $\sum_{j \in U}(m_{ij} - |\mathcal{J}''(I_j) \cap \mathcal{J}_i|)|I_j|s_i$, the expression in (3.2).

So suppose that (3.4) is violated for some $j \in U$. Then $|\mathcal{J}'(I_j) \cap \mathcal{J}_i| > m_{ij} - |\mathcal{J}''(I_j) \cap \mathcal{J}_i|$ and the number of jobs of $\mathcal{J}_i$ that are active in I_j is strictly larger than m_{ij}. Hence m_{ij} is the total number of processors not yet occupied by $\mathcal{J}_i \cup \ldots \cup \mathcal{J}_{i-1}$. As the number of jobs of $\mathcal{J}$ active in I_j is at least as large as the corresponding number of jobs of $\mathcal{J}_i$, we have $m_j = m_{ij}$. Since $|\mathcal{J}''(I_j)| \geq |\mathcal{J}''(I_j) \cap \mathcal{J}_i|$ we have $m_j - |\mathcal{J}''(I_j)| \leq m_{ij} - |\mathcal{J}''(I_j) \cap \mathcal{J}_i|$, which contradicts the assumption that (3.4) was violated. □

Lemma 17. *When phase i ends, $\mathcal{J} = \mathcal{J}_i$.*

Proof. When phase i ends, consider set $\mathcal{J}$ and the flow in the current graph $G = (\mathcal{J}, \vec{m}, s)$. Again, for any edge e let $f(e)$ be the amount of flow along e. Since the value of the computed flow is equal to F_G we have $f(u_0, u_k) = v_k/s$, for any $J_k \in \mathcal{J}$. By Lemma 16, $\mathcal{J}_i \subseteq \mathcal{J}$. So suppose $\mathcal{J}_i \neq \mathcal{J}$, which implies $\mathcal{J}_i \subset \mathcal{J}$. We prove that in this case the speed s_i used for $\mathcal{J}_i$ in an optimal schedule satisfies $s_i \leq s$ and then derive a contradiction.

In the graph $G = (\mathcal{J}, \vec{m}, s)$ we remove the flow associated with any $J_k \in \mathcal{J} \setminus \mathcal{J}_i$. More specifically, for any such J_k, we remove $f(u_k, w_j)$ units of flow along the path (u_0, u_k), (u_k, w_j) and (w_j, w_0) until the flow along (u_0, u_k) is zero. Let $f'(w_j, w_0)$ be the new amount of flow along (w_j, w_0) after the modifications. Then for any interval I_j, at least $\lceil f'(w_j, w_0)/|I_j| \rceil$ jobs of $\mathcal{J}_i$ are active in I_j because for each job at most $|I_j|$ units of flow are routed into w_j. Also, at least $\lceil f'(w_j, w_0)/|I_j| \rceil$ processors are available because $m_j = f(w_j, w_0)/|I_j|$ processors are available in I_j. Hence

$$s_i \le \frac{\sum_{J_k \in \mathcal{J}_i} v_k}{\sum_{1 \le j < |\mathcal{I}|} \lceil f'(w_j, w_0)/|I_j| \rceil |I_j|}.$$

The latter expression is upper bounded by s because

$$\begin{aligned} \sum_{J_k \in \mathcal{J}_i} v_k &= s \sum_{J_k \in \mathcal{J}_i} v_k/s \;=\; s \sum_{J_k \in \mathcal{J}_i} f(u_0, u_k) \\ &= s \sum_{1 \le j < |\mathcal{I}|} f'(w_j, w_0) \le s \sum_{1 \le j < |\mathcal{I}|} \lceil f'(w_j, w_0)/|I_j| \rceil |I_j|. \end{aligned}$$

Hence $\mathcal{J}_i$ is processed at speed $s_i \le s$ while jobs of $\mathcal{J} \setminus \mathcal{J}_i$ are processed at speed s' with $s' < s_i \le s$. However, this is impossible because the minimum average speed to process the jobs of $\mathcal{J}$ is $s = \sum_{J_k \in \mathcal{J}} v_k / \sum_{1 \le j < |\mathcal{I}|} m_j |I_j|$. $\square$

We conclude with the following theorem.

Theorem 8. *An optimal schedule can be computed in polynomial time using combinatorial techniques.*

3.2 Online algorithms

We present adaptions and extensions of the single processor algorithms *Optimal Available* and *Average Rate* [55]. Throughout this section, we consider power functions of the form $P(s) = s^\alpha$, where $\alpha > 1$.

3.2.1 Algorithm Optimal Available

For single processor environments, *Optimal Available (OA)* works as follows: Whenever a new job arrives, *OA* computes an optimal schedule for the currently available, unfinished jobs. While it is straightforward to extend this strategy to parallel processing environments, the corresponding competitive analysis becomes considerably more involved. We have to prove a series of properties of the online algorithm's schedule and analyze how a schedule changes in response to the arrival of a new job. Our algorithm for parallel processors works as follows.

Algorithm OA(m): Whenever a new job arrives, compute an optimal schedule for the currently available unfinished jobs. This can be done using the algorithm of Section 3.1.

At any given time t_0, let $S_{OA(m)}$ denote the schedule of *OA*(m) for the currently available unfinished jobs. Let $J_1, \ldots, J_n$ be the corresponding jobs, which have arrived by time t_0 but are still unfinished. Let v_i be the remaining processing volume of J_i, $1 \leq i \leq n$. Since $J_1, \ldots, J_n$ are available at time t_0 we can ignore release times and only have to consider the deadlines $d_1, \ldots, d_n$. Let $\mathcal{I} = \{t_0\} \cup \{d_i \mid 1 \leq i \leq n\}$ be the set of relevant time points and $\tau_1 < \ldots < \tau_{|\mathcal{I}|}$ be the sorted order of the elements of $\mathcal{I}$, where $|\mathcal{I}| \leq n+1$. In the time horizon $[\tau_1, \tau_{|\mathcal{I}|})$, the j-th interval is $I_j = [\tau_j, \tau_{j+1})$, where $1 \leq j < |\mathcal{I}|$. Schedule $S_{OA(m)}$ is an optimal schedule for the current jobs $J_1, \ldots, J_n$; otherwise when last computing a schedule, *OA*(m) would have obtained a better solution for time horizon $[\tau_1, \tau_{|\mathcal{I}|})$. By Lemma 13 we always assume that in $S_{OA(m)}$ each job is processed at a constant speed.

For the given time t_0, let $\mathcal{S}_{OPT}$ denote the set of optimal schedules for the unfinished jobs $J_1, \ldots, J_n$. We have $S_{OA(m)} \in \mathcal{S}_{OPT}$. Since $P(s) = s^\alpha$, there holds $\mathcal{S}_{OPT} = \mathcal{S}'_{OPT}$. Recall that the latter set was introduced in the beginning of Section 3.1 as the subset of optimal schedules that minimize energy consumption according to the power function $P(s) = s^\alpha$. Hence $S_{OA(m)} \in \mathcal{S}'_{OPT}$ and, by Lemma 14, in each interval I_j of $S_{OA(m)}$ we can rearrange the schedule such that each processor uses a fixed, non-varying speed in I_j, $1 \leq j < |\mathcal{I}|$. In the following we restrict ourselves to schedules $S_{OA(m)}$ satisfying this property. Such schedules are feasible and incur a minimum energy. Let $s_{l,j}$ be the speed used by processor l in I_j, where $1 \leq l \leq m$ and $1 \leq j < |\mathcal{I}|$. Also note that Lemma 15 holds.

The next Lemma 18 states that we can modify $S_{OA(m)}$ even further so that on each processor the speed levels used form a non-increasing sequence. The modification is obtained by simply permuting within each interval I_j the schedules on the m processors, $1 \leq j < |\mathcal{I}|$. Hence feasibility and energy consumption of the overall schedule are not affected. Throughout the analysis we always consider schedules $S_{OA(m)}$ that also fulfill this additional property. This will be essential in the proofs of the subsequent Lemmas 19–21.

Lemma 18. *Given $S_{OA(m)}$, in each interval I_j, $1 \leq j < |\mathcal{I}|$, we can permute the schedules on the m processors such that the following property holds. For any processor l, $1 \leq l \leq m$, inequality $s_{l,j} \geq s_{l,j+1}$ is satisfied for $j = 1, \ldots, |\mathcal{I}| - 2$.*

Proof. In each interval I_j, $1 \leq j < |\mathcal{I}|$, we simply sort the schedules on the m

processors in order of non-increasing speeds such that $s_{1,j} \geq \ldots \geq s_{m,j}$. Consider an arbitrary j, $1 \leq j \leq |\mathcal{I}| - 2$, and suppose that the desired property does not hold for all l, $1 \leq l \leq m$. Then let l_0 be the smallest index such that $s_{l_0,j} < s_{l_0,j+1}$. In I_{j+1} the first l_0 processors, running at speeds $s_{1,j+1} \geq \ldots \geq s_{l_0,j+1}$, execute a set $\mathcal{J}'(I_{j+1})$ of at least l_0 different jobs in the interval because a job cannot be executed in parallel on two or more processors. The jobs of $\mathcal{J}'(I_{j+1})$ can only be processed on the first $l_0 - 1$ processors in I_j because in $S_{OA(m)}$ each job is processed at a constant speed and the last $m - l_0 + 1$ processors in I_j use speeds which are strictly smaller than $s_{1,j+1}, \ldots, s_{l_0,j+1}$. Consider a time window $W = [\tau_j, \tau_j + \delta)$ with $\delta > 0$ in I_j such that each of the first l_0 processors does not change the job it executes in W. Among the at least l_0 jobs of $\mathcal{J}'(I_{j+1})$ there must exist a J_i which is not processed in W because jobs of $\mathcal{J}'(I_{j+1})$ are not executed on the $m - l_0 + 1$ last processors in I_j and the first $l_0 - 1$ processors handle at most $l_0 - 1$ different jobs of $\mathcal{J}'(I_{j+1})$ in W. Suppose that J_i is processed for δ_i time units in I_{j+1} using speed $s(J_i)$. Set $\delta' = \min\{\delta, \delta_i\}$ and $\epsilon = (s_{l_0,j+1} - s_{l_0,j})/2$. We now modify the schedule as follows. In I_{j+1} for δ' time units where J_i is processed we reduce the speed by ϵ. In I_j, in the time window $W' = [\tau_j, \tau_j + \delta')$ we increase the speed of processor l_0 by ϵ and use the extra processing capacity of $\delta'\epsilon$ units to process $\delta'\epsilon$ units of J_i. We obtain a feasible schedule. Since $s(J_i) - \epsilon \geq s_{j+1,l_0} - \epsilon = s_{j,l_0} + \epsilon$, by the convexity of the power function $P(s) = s^\alpha$ the modified schedule has a strictly smaller energy consumption. This is a contradiction to the fact that the original schedule was optimal. □

We investigate the event that a new job J_{n+1} arrives at time t_0. As usual d_{n+1} and v_{n+1} are the deadline and the processing volume of J_{n+1}. Let $S_{OA(m)}$ be the schedule of *OA*(m) immediately before the arrival of J_{n+1}, and let $S'_{OA(m)}$ be the schedule immediately after the arrival when *OA*(m) has just computed a new schedule. We present two lemmas that relate the two schedules. Loosely speaking we prove that processor speeds can only increase. The speed at which a job J_k, $1 \leq k \leq n$, is processed in $S'_{OA(m)}$ is at least as high as the corresponding speed in $S_{OA(m)}$. Furthermore, at any time $t \geq t_0$, the minimum processor speed in $S'_{OA(m)}$ is at least as high as that in $S_{OA(m)}$.

Schedules $S_{OA(m)}$ and $S'_{OA(m)}$ are optimal schedules for the respective workloads. Recall that they process jobs at constant speeds. Let $s(J_k)$ be the speed at which J_k is processed in $S_{OA(m)}$, $1 \leq k \leq n$. Let $s'(J_k)$ be the speed used for J_k in $S'_{OA(m)}$, $1 \leq k \leq n+1$.

Lemma 19. *There holds* $s'(J_k) \geq s(J_k)$, *for any* $1 \leq k \leq n$.

Proof. In $S_{OA(m)}$ let $s_1 > s_2 > \ldots > s_p$ be the speeds used in the schedule and let $\mathcal{J}_1, \ldots, \mathcal{J}_p$ be the associated job sets, i.e. the jobs of $\mathcal{J}_i$ are processed at speed s_i, $1 \leq i \leq p$. In the following we assume that the statement of the lemma does not hold and derive a contradiction. So let i be the smallest index such that $\mathcal{J}_i$ contains a job J_k with $s'(J_k) < s(J_k) = s_i$. We partition $\mathcal{J}_i$ into two sets $\mathcal{J}_{i1}$ and $\mathcal{J}_{i2}$ such that all $J_k \in \mathcal{J}_{i1}$ satisfy $s'(J_k) \geq s_i$, as desired. For each job $J_k \in \mathcal{J}_{i2}$ we have $s'(J_k) < s_i$. By the choice of i, any job $J_k \in \mathcal{J}_1 \cup \ldots \cup \mathcal{J}_{i-1}$ also fulfills the desired property $s'(J_k) \geq s(J_k)$.

Let t_1 be the first point of time when the processing of any job of $\mathcal{J}_{i2}$ starts in $S'_{OA(m)}$. Let P be a processor executing such a job at time t_1. We first argue that any $J_k \in \mathcal{J}_1 \cup \ldots \cup \mathcal{J}_{i-1} \cup \mathcal{J}_{i1}$ that is active at time $t \geq t_1$, i.e. that satisfies $d_k > t_1$, is scheduled at all times throughout $[t_1, d_k)$: So suppose J_k were not executed on any of the processors in a time window $W \subseteq [t_1, d_k)$ and let $\delta > 0$ be the length of W. Determine a δ' with $0 < \delta' \leq \delta$ such that during the first δ' time units of W processor P does not change the job it executes. We set $\delta_k = \min\{\delta', v_k/s'(J_k)\}$. Here $v_k/s'(J_k)$ is the total time for which J_k is scheduled in $S'_{OA(m)}$. Let $s(P, W)$ be the maximum speed used by processor P within W and set $\epsilon = (s'(J_k) - s(P, W))/2$. The value of ϵ is strictly positive because $s'(J_k) \geq s_i > s(P, W)$. The last inequality holds because at time t_1 processor P executes a job of $\mathcal{J}_{i2}$ at a speed that is strictly smaller than s_i and we consider schedules that satisfy the property of Lemma 18, i.e. on each processor the speed levels over time are non-increasing. We now modify $S'_{OA(m)}$ as follows. During any δ_k time units where J_k is processed we reduce the speed by ϵ. During the first δ_k time units of W we increase the speed by ϵ and use the extra processing capacity of $\delta_k\epsilon$ units to execute $\delta_k\epsilon$ units of J_k. Hence we obtain a new feasible schedule. Since $s'(J_k) - \epsilon = s(P, W) + \epsilon$, by the convexity of the power function $P(s) = s^\alpha$, the schedule has a strictly smaller energy consumption, contradicting the fact that $S'_{OA(m)}$ is an optimal schedule for the given workload.

Next consider any fixed time $t \geq t_1$. In $S_{OA(m)}$ let m_{i2} be the number of processors that execute jobs of $\mathcal{J}_{i2}$ at time t. Similarly, in $S'_{OA(m)}$ let m'_{i2} be the number of processors that execute jobs of $\mathcal{J}_{i2}$ at time t. We will prove $m_{i2} \geq m'_{i2}$. Let n_l, where $1 \leq l \leq i-1$, be the number of jobs J_k of $\mathcal{J}_l$ that are active at time t, i.e. that satisfy $d_k > t$. Furthermore, let n_{i1} and n_{i2} be the number of jobs of $\mathcal{J}_{i1}$ and $\mathcal{J}_{i2}$, respectively, that are active at time t. As shown in the last paragraph, all jobs of $\mathcal{J}_1 \cup \ldots \cup \mathcal{J}_{i-1} \cup \mathcal{J}_{i1}$ that are active at time t are also processed at this time in $S'_{OA(m)}$. Hence

$$m'_{i2} \leq m - \sum_{l=1}^{i-1} n_l - n_{i1}. \tag{3.5}$$

Strict inequality holds, for instance, if the new job J_{n+1} or a job of $\mathcal{J}_{i+1} \cup \ldots \cup \mathcal{J}_p$ is scheduled at time t. Let m_l be the number of processors executing jobs of $\mathcal{J}_l$ in $S_{OA(m)}$, where $1 \leq l \leq i$. By Lemma 15, $m_i = \min\{n_i, m - \sum_{l=1}^{i-1} m_l\}$. We remark that Lemma 15 is formulated with respect to an interval I_j. We can apply the statement simply by considering the interval containing t. Since $m_l \leq n_l$, for

$1 \leq l \leq i-1$, we obtain

$$m_i \geq \min\{n_i, m - \sum_{l=1}^{i-1} n_l\}. \tag{3.6}$$

If the latter minimum is given by the term $m - \sum_{l=1}^{i-1} n_l$, then in (3.5) we obtain $m'_{i2} \leq m - \sum_{l=1}^{i-1} n_l - n_{i1} \leq m_i - n_{i1} \leq m_{i2}$. The last inequality holds because m_{i2} cannot be smaller than m_i, the number of machines used by jobs of $\mathcal{J}_i$, minus the number of jobs of $\mathcal{J}_{i1}$ active at time t. Hence $m'_{i2} \leq m_{i2}$, as desired. If the minimum in (3.6) is given by n_i, then since $m_{i2} \geq m_i - n_{i1}$, we have $m_{i2} \geq n_i - n_{i1}$. As $n_i - n_{i1} = n_{i2} \geq m'_{i2}$ we conclude again $m_{i2} \geq m'_{i2}$.

Finally let T' be the total processing time used for jobs of $\mathcal{J}_{i2}$ in $S'_{OA(m)}$. This value can be determined as follows. The time horizon of $S'_{OA(m)}$ is partitioned into maximal intervals such that the number of processors handling jobs of $\mathcal{J}_{i2}$ does not change within an interval. For each such interval, the length of the interval is multiplied by the number of processors executing jobs of $\mathcal{J}_{i2}$. Value T' is simply the sum of all these products. Similarly, let T be the total processing time used for jobs of $\mathcal{J}_{i2}$ in $S_{OA(m)}$. At any time t we have $m_{i2} \geq m'_{i2}$. This was shown above for times $t \geq t_1$. It trivially holds for times t with $t_0 \leq t < t_1$ because no jobs of $\mathcal{J}_{i2}$ are processed in $S'_{OA(m)}$ at that time. Hence the number of processors handling jobs of $\mathcal{J}_{i2}$ in $S_{OA(m)}$ is always at least as high as the corresponding number in $S_{OA(m)}$. This implies $T \geq T'$. The total processing volume $\sum_{J_k \in \mathcal{J}_{i2}} v_k$ of jobs in $\mathcal{J}_{i2}$ fills T' time units using a speed or speeds of $s < s_i$ in $S'_{OA(m)}$. Thus using a speed of s_i the processing volume is not sufficient to fill $T > T'$ time units in $S_{OA(m)}$. We obtain a contradiction to the fact that jobs of $\mathcal{J}_i$ are processed at constant speed s_i in $S_{OA(m)}$. □

Lemma 20. *At any time t, where $t \geq t_0$, the minimum speed used by the m processors at time t in $S'_{OA(m)}$ is at least as high as the minimum speed used on the m processors at time t in $S_{OA(m)}$.*

Proof. Consider an arbitrary time $t \geq t_0$. Let $s_{\min}$ be the minimum speed on the m processors in $S_{OA(m)}$ at time t. Let $s'_{\min}$ be the corresponding minimum speed in $S'_{OA(m)}$ at time t. If $s_{\min} = 0$, then the lemma trivially holds. So suppose $s_{\min} > 0$ and assume that the statement of the lemma does not hold, i.e. $s'_{\min} < s_{\min}$. Since $s_{\min} > 0$, m different jobs are processed at time t in $S_{OA(m)}$. By Lemma 19, for any job, the speed at which it is processed in $S'_{OA(m)}$ is at least as high as the corresponding speed in $S_{OA(m)}$. Hence among the m jobs processed in $S_{OA(m)}$ at time t there must exist at least one job J_k that is not executed in $S'_{OA(m)}$ at time t. For this job we have $d_k > t$ because J_k is available for processing and hence active at time t. Determine a $\delta_k > 0$ with $\delta_k \leq \min\{d_k - t, v_k/s'(J_k)\}$ such that J_k is not processed in the interval $[t, t+\delta_k)$ in $S'_{OA(m)}$. Choose a δ with $0 < \delta \leq \delta_k$ such that in $S'_{OA(m)}$ the last processor m, running at minimum speed $s'_{\min}$ at time t, does not change the job it executes within time window $W = [t, t+\delta)$. Throughout W a speed of $s'_{\min}$ is used on processor m. Set $\epsilon = (s_{\min} - s'_{\min})/2$, which is a strictly positive value.

We now modify $S'_{OA(m)}$. During δ time units in which J_k is processed we reduce the speed by ϵ. These time units exist because $\delta \leq v_k/s'(J_k)$. In the time window W we increase the speed on the last processor m by ϵ and use the extra processing capacity of $\delta\epsilon$ units to process $\delta\epsilon$ units of J_k. Since $\delta \leq d_k - t$ job J_k finishes by its deadline, and we obtain a feasible schedule. Throughout W the newly set speed on the last processor m is $s'_{\min} + \epsilon$. Finally notice $s'_{\min} + \epsilon = s_{\min} - \epsilon \leq s(J_k) - \epsilon \leq s'(J_k) - \epsilon$. The last inequality follows from Lemma 19. Hence $s'_{\min} + \epsilon \leq s'(J_k) - \epsilon$, and by the convexity of the power function $P(s) = s^\alpha$, we obtain a schedule with a strictly smaller power consumption. This is a contradiction to the fact that $S'_{OA(m)}$ is optimal. □

We finally need a lemma that specifies the minimum processor speed for a time window if a job finishes at some time strictly before its deadline.

Lemma 21. *If in $S_{OA(m)}$ a job J_k finishes at some time t with $t < d_k$, then throughout $[t, d_k)$ the minimum speed on the m processors is at least $s(J_k)$.*

Proof. Consider a job J_k that finishes at time $t < d_k$. Suppose that the lemma does not hold and let t' with $t \leq t' < d_k$ be a time such that the minimum speed on the m processors is strictly smaller than $s(J_k)$. Recall that we consider schedules satisfying the property of Lemma 18, i.e. on any processor the speed levels form a non-increasing sequence. Hence within $W = [t', d_k)$ there exists a processor P whose speed throughout W is upper bounded by some $s < s(J_k)$. Choose a positive δ with $\delta \leq \min\{d_k - t', v_k/s(J_k)\}$ such that processor P does not switch jobs in $W' = [t', t'+\delta)$. Set $\epsilon = (s(J_k) - s)/2$. We modify the schedule as follows. During δ time units where J_k is processed we reduce the speed by ϵ. In W' we increase the speed of processor P by ϵ and use the extra processing capacity to finish the missing $\delta\epsilon$ processing units of J_k. The resulting schedule is feasible and incurs a strictly smaller energy consumption. Hence we obtain a contradiction. □

We next turn to the competitive analysis of *OA*(m). We use a potential function that is inspired by a potential used by Bansal et al. [19]. We first define our modified potential function. At any time t consider the schedule of $S_{OA(m)}$. Let $s_1 > \ldots > s_p$ be the speeds used in the schedule and let $\mathcal{J}_1, \ldots, \mathcal{J}_p$ be the associated job sets, i.e. jobs of $\mathcal{J}_i$ are processed at speed s_i. Let $V_{OA(m)}(i)$ be the total remaining processing volume of jobs $J_k \in \mathcal{J}_i$ in *OA*(m)'s schedule $1 \leq i \leq p$.

Similarly, let $V_{OPT}(i)$ be the total remaining processing volume that *OPT* has to finish for jobs $J_k \in \mathcal{J}_i$, $1 \leq i \leq p$. We remark that, for a job $J_k \in \mathcal{J}_i$, *OPT*'s remaining work might be larger than that of *OA*(m) if *OPT* has completed less work on J_k so far. Sets $\mathcal{J}_1, \dots, \mathcal{J}_p$ represent the jobs that *OA*(m) has not yet finished. Let $\mathcal{J}'$ be the set of jobs that are finished by *OA*(m) but still unfinished by *OPT*. Obviously, these jobs have a deadline after the current time t. If $\mathcal{J}' \neq \emptyset$, then for any job of $\mathcal{J}'$ consider the speed *OA*(m) used when last processing the job. Partition $\mathcal{J}'$ according to these speeds, i.e. $\mathcal{J}'_i$ consists of those jobs that *OA*(m) executed last at speed s'_i, where $1 \leq i \leq p'$ for some positive p'. Let $V'_{OPT}(i)$ be the total remaining processing volume *OPT* has to finish on jobs of $\mathcal{J}'_i$, $1 \leq i \leq p'$. If $\mathcal{J}' = \emptyset$, we simply set $s'_1 = 0$ and $\mathcal{J}'_1 = \emptyset$. The potential Φ is defined as follows.

$$\Phi = \alpha \sum_{i \geq 1} (s_i)^{\alpha-1}(V_{OA(m)}(i) - \alpha V_{OPT}(i)) - \alpha^2 \sum_{i \geq 1} (s'_i)^{\alpha-1} V'_{OPT}(i)$$

Compared to the potential function used to analyze *OA* in the single processor setting, our function here consists of a second term representing *OPT*'s unfinished work. This second term is essential to establish the competitiveness of α^α and, in particular, is crucially used in the analysis of the working case (see below) where we have to consider m processor pairs.

At any time t let $s_{OA(m),1}(t) \geq \dots \geq s_{OA(m),m}(t)$ and $s_{OPT,1}(t) \geq \dots \geq s_{OPT,m}(t)$ be the speeds used by *OA*(m) and *OPT* on the m processors, respectively. In the remainder of this section we will prove the following properties.

(a) Whenever a new job arrives or a job is finished by *OA*(m) or *OPT*, the potential does not increase.

(b) At all other times

$$\sum_{l=1}^{m} (s_{OA(m),l}(t))^\alpha - \alpha^\alpha \sum_{l=1}^{m} (s_{OPT,l}(t))^\alpha + \frac{d\Phi(t)}{dt} \leq 0. \tag{3.7}$$

Integrating over all times we obtain that *OA*(m) is α^α-competitive.

Working case: We prove property (b). So let t be any time when the algorithms are working on jobs. In a first step we match the processors of *OA*(m) to those of *OPT*. This matching is constructed as follows. First, processors handling the same jobs are paired. More specifically, if a job J_k is executed on a processor l in *OA*(m)'s schedule and on processor l' in *OPT*'s schedule, then we match the two processors l and l'. All remaining processors of *OA*(m) and *OPT*, handling disjoint job sets, are matched in an arbitrary way.

For each matched processor pair l, l', we consider its contribution in (3.7) and prove that the contribution is non-positive. Summing over all pairs l, l', this establishes (3.7). If the speed of processor l' in *OPT*'s schedule is 0, then the analysis is simple: If the speed on processor l in *OA*(m) schedule is also 0, then the contribution of the processor pair l, l' in (3.7) is 0. If the speed on processor l is positive, then *OA*(m) executes a job on the processor. Suppose that the job is contained in set $\mathcal{J}_i$. Then in Φ the value of $V_{OA(m)}(i)$ decreases at rate s_i and the contribution of the processor pair l, l' in (3.7) is $s_i^\alpha - \alpha(s_i)^{\alpha-1}s_i < 0$.

Hence in the following we assume that the speed of processor l' in *OPT*'s schedule is positive. We show that *OA*(m) also executes a job J_k on its processor l and that the contribution of the processor pair l, l' is upper bounded by

$$(1-\alpha)s_i^\alpha + \alpha^2 s_i^{\alpha-1}s_{OPT,l'}(t) - \alpha^\alpha(s_{OPT,l'}(t))^\alpha, \tag{3.8}$$

where i is the index such that $J_k \in \mathcal{J}_i$. As in [19] one can then show the non-positivity of this expression. First consider the case that both processors l and l' handle the same job J_k. Let $\mathcal{J}_i$ be the set containing J_k. Then $V_{OA(m)}(i)$ decreases at rate s_i and $V_{OPT}(i)$ decreases at rate $s_{OPT,l'}(t)$. Hence the contribution of the processor pair in (3.7) is $s_i^\alpha - \alpha^\alpha(s_{OPT,l'}(t))^\alpha + (-\alpha(s_i)^{\alpha-1}s_i + \alpha^2(s_i)^{\alpha-1}s_{OPT,l'}(t)) = (1-\alpha)s_i^\alpha + \alpha^2(s_i)^{\alpha-1}s_{OPT,l'}(t) - \alpha^\alpha(s_{OPT,l'}(t))^\alpha$, as stated in (3.8).

Next consider the case that the job $J_{k'}$ executed on processor l' in *OPT*'s schedule is not executed by *OA*(m) on processor l and hence is not executed on any of its processors at time t. If $J_{k'}$ is still unfinished by *OA*(m), then let $\mathcal{J}_{i'}$ be the set containing $J_{k'}$. Recall that *OA*(m)'s schedule is optimal. Hence, by Lemma 15, it always processes the jobs from the smallest indexed sets $\mathcal{J}_i$, which in turn use the highest speeds s_i. Thus, if $J_{k'}$ is not executed by *OA*(m), then the schedule currently processes m jobs, each of which is contained in $\mathcal{J}_1 \cup \ldots \cup \mathcal{J}_{i'-1}$ or contained in $\mathcal{J}_{i'}$ but is different from $J_{k'}$. Let J_k be the job executed by *OA*(m) on processor l and let $\mathcal{J}_i$ be the job set containing J_k. Then $i \le i'$ and $s_i \ge s_{i'}$. In Φ the term $V_{OA(m)}(i)$ decreases at rate s_i while the term $V_{OPT}(i')$ decreases at rate $s_{OPT,l'}(t)$. Hence the contribution of the processor pair l, l' in (3.7) is $s_i^\alpha - \alpha^\alpha(s_{OPT,l'}(t))^\alpha + (-\alpha(s_i)^{\alpha-1}s_i + \alpha^2(s_{i'})^{\alpha-1}s_{OPT,l'}(t)) \le (1-\alpha)s_i^\alpha + \alpha^2(s_i)^{\alpha-1}s_{OPT,l'}(t) - \alpha^\alpha(s_{OPT,l'}(t))^\alpha$, which is the bound of (3.8).

It remains to consider the case that $J_{k'}$ is already finished by *OA*(m). Then $J_{k'}$ is contained in the set $\mathcal{J}'$ of jobs that are finished by *OA*(m) but unfinished by *OPT*. Let $\mathcal{J}'_{i'}$ be the set containing $J_{k'}$. In Φ the term $V'_{OPT}(i')$ decreases at rate $s_{OPT,l'}(t)$. When *OA*(m) finished $J_{k'}$ in its schedule, it used speed $s'_{i'}$. By Lemma 21, in the former schedule of *OA*(m), the minimum processor speed throughout the time window until $d_{k'}$ was always at least $s'_{i'}$. This time window contains the current time, i.e. $t < d_{k'}$. Since the completion of $J_{k'}$ *OA*(m) might have computed new schedules in response to the arrival of new jobs but, by Lemma 20, the minimum processor speed at any time can only increase. Hence in the current schedule of *OA*(m) and at the current time t the minimum processor speed is at least $s'_{i'}$. Thus on its processor l *OA*(m) uses a speed of at least $s'_{i'} > 0$. Hence *OA*(m) executes a job J_k that belongs to, say, set $\mathcal{J}_i$. We have $s_i \geq s'_{i'}$. In Φ the term $V_{OA(m)}(i)$ decreases at rate s_i. We conclude again that the contribution of the processor pair l, l' is $s_i^\alpha - \alpha^\alpha(s_{OPT,l'}(t))^\alpha + (-\alpha(s_i)^\alpha + \alpha^2(s'_{i'})^{\alpha-1}s_{OPT,l'}(t)) \leq (1-\alpha)s_i^\alpha + \alpha^2(s_i)^{\alpha-1}s_{OPT,l'}(t) - \alpha^\alpha(s_{OPT,l'}(t))^\alpha$, which is again the bound of (3.8).

Completion/Arrival case: When *OPT* finishes a job J_k the potential does not change because in the terms $V_{OPT}(i)$ or $V_{OPT}(i')$ the remaining processing volume of J_k simply goes to 0. Similarly if *OA*(m) completes a job already finished by *OPT*, the potential does not change. If *OA*(m) completes a job J_k not yet finished by *OPT*, then let $\mathcal{J}_i$ be the set containing J_k immediately before completion. In the first term of Φ, $V_{OPT}(i)$ increases by $\alpha^2(s_i)^{\alpha-1}v_k$, where v_k is the remaining work of J_k to be done by *OPT*. However, at the same time, the second term of Φ decreases by exactly $\alpha^2(s_i)^{\alpha-1}v_k$ because J_k opens or joins a set $\mathcal{J}'_{i'}$ with $s'_{i'} = s_i$.

When a new job J_{n+1} arrives, the structure of the analysis is similar to that in the single processor case, see [19]. We imagine that a job J_{n+1} of processing volume 0 arrives. Then we increase the processing volume to its true size. However, in the multi-processor setting, we have to study how the schedule of *OA*(m) changes if the processing volume of J_{n+1} increases from a value v_{n+1} to $v'_{n+1} = v_{n+1} + \epsilon$, for some $\epsilon > 0$. Let $S_{OA(m)}$ and $S'_{OA(m)}$ be the schedules of *OA*(m) before and after the increase, respectively. For any job J_k, $1 \leq k \leq n+1$, let $s(J_k)$ and $s'(J_k)$ be the speeds used for J_k in $S_{OA(m)}$ and $S'_{OA(m)}$, respectively. The next lemma states that the speeds at which jobs are processed does not decrease. The proof is identical to that of Lemma 19.

Lemma 22. *There holds* $s'(J_k) \geq s(J_k)$*, for any* $1 \leq k \leq n+1$.

In $S_{OA(m)}$, let $\mathcal{J}_{i_0}$ be the job set containing J_{n+1}, i.e. $J_{n+1} \in \mathcal{J}_{i_0}$, where $1 \leq i_0 \leq p$. The following lemma implies that the speeds of jobs $J_k \in \mathcal{J}_{i_0+1} \cup \ldots \cup \mathcal{J}_p$ do not change when increasing the processing volume of J_{n+1}.

Lemma 23. *There holds* $s'(J_k) = s(J_k)$*, for any* $J_k \in \mathcal{J}_{i_0+1} \cup \ldots \cup J_p$.

Proof. We first analyze $S_{OA(m)}$ and $S'_{OA(m)}$ with respect to the times when jobs of $\mathcal{J}_{i_0+1} \cup \ldots \cup \mathcal{J}_p$ are processed. Consider any time $t \geq t_0$ in the scheduling horizon, where t_0 is again the current time. Let m_1 be the number of processors executing jobs of $\mathcal{J}_1 \cup \ldots \cup \mathcal{J}_{i_0}$ in $S_{OA(m)}$ at time t, and let m_2 be the number of processors executing jobs of $\mathcal{J}_{i_0+1} \cup \ldots \cup \mathcal{J}_p$ in $S_{OA(m)}$ at time t. Values m'_1 and m'_2 are defined analogously with respect to $S'_{OA(m)}$. In the following we will prove $m'_2 \geq m_2$.

If $m_2 = 0$, there is nothing to show. So suppose $m_2 > 0$. By Lemma 15 exactly m_1 jobs of $\mathcal{J}_1 \cup \ldots \cup \mathcal{J}_{i_0}$ are active at time t, i.e. have a deadline after time

t; if there were more active jobs, *OA*(m) would have assigned more processors to the jobs of $\mathcal{J}_1 \cup \ldots \cup \mathcal{J}_{i_0}$ because they are executed at higher speeds than the jobs of $\mathcal{J}_{i_0+1} \cup \ldots \cup \mathcal{J}_p$. We next assume $m_2' < m_2$ and derive a contradiction. We distinguish two cases depending on whether or not $S'_{OA(m)}$ has an idle processor at time t.

If there is no idle processor in $S'_{OA(m)}$ at time t, then $m_1' = m - m_2' > m - m_2 \geq m_1$ jobs of $\mathcal{J}_1 \cup \ldots \cup \mathcal{J}_{i_0}$ must be executed in $S'_{OA(m)}$ at time t. However, this is impossible because only m_1 jobs of this union of sets are active and hence available for processing at time t. On the other hand, if there is a processor P that is idle in $S'_{OA(m)}$ at time t, then there must exist a time window W containing time t such that P is idle throughout W. Moreover, since $m_2' < m_2$, there must exist a job $J_k \in \mathcal{J}_{i_0+1} \cup \ldots \cup \mathcal{J}_p$ that is active but not processed at time t. Hence, within W we can select a time window $W' \subseteq W$ of length δ, where $\delta \leq v(J_k)/s'(J_k)$, such that J_k is not executed on any processor in $S'_{OA(m)}$ throughout W'. We now modify $S'_{OA(m)}$ as follows. For any δ time units where J_k is scheduled, we reduce the speed to $s'(J_k)/2$. On processor P we execute J_k at speed $s'(J_k)/2$ for δ time units in W'. By convexity of the power function, we obtain a better schedule with a strictly smaller energy consumption, contradicting the fact that $S'_{OA(m)}$ is an optimal schedule.

Let T be the total time used to process jobs of $\mathcal{J}_{i_0+1} \cup \ldots \cup \mathcal{J}_p$ in $S_{OA(m)}$. As in the proof of Lemma 19, this time is computed as follows. We partition the scheduling horizon of $S_{OA(m)}$ into maximal intervals such that within each interval the number of processors executing jobs of $\mathcal{J}_{i_0+1} \cup \ldots \cup \mathcal{J}_p$ does not change. The length of each interval is multiplied by the number of processors handling jobs of $\mathcal{J}_{i_0+1} \cup \ldots \cup \mathcal{J}_p$. Then T is simply the sum of these products. Analogously, let T' be the total time used to process jobs of $\mathcal{J}_{i_0+1} \cup \ldots \cup \mathcal{J}_p$ in $S'_{OA(m)}$. In the last paragraph we showed that at any time t, $m_2' \geq m_2$. Hence $T' \geq T$.

We are now ready to finish the proof of our lemma. By Lemma 22 we have $s'(J_k) \geq s(J_k)$ for any $J_k \in \mathcal{J}_{i_0+1} \cup \ldots \cup \mathcal{J}_p$. Suppose that in the latter set there exists a job J_{k_0} with $s'(J_{k_0}) > s(J_{k_0})$. In $S_{OA(m)}$ the jobs of $\mathcal{J}_{i_0+1} \cup \ldots \cup \mathcal{J}_p$ are processed for T time units. The total processing volume of these jobs cannot fill $T' \geq T$ time units if J_{k_0} is processed at a strictly higher speed $s'(J_{k_0}) > s(J_{k_0})$ while all other jobs $J_k \in \mathcal{J}_{i_0+1} \cup \ldots \cup \mathcal{J}_p$ are processed at speed $s'(J_k) \geq s(J_k)$. We obtain a contradiction. □

In the remainder of this section we describe the change in potential arising in response to the arrival of a new job J_{n+1}. If the deadline d_{n+1} does not coincide with the deadline of a previous job J_i, $1 \leq i \leq n$, we have to update the set of intervals I_j, $1 \leq j < |\mathcal{I}|$. We add d_{n+1} to $\mathcal{I}$. If d_{n+1} is the last deadline of any of the jobs, we introduce a new interval $[\tau_{\max}, d_{n+1})$, where $\tau_{\max} = \max_{1\leq i \leq n} d_i$; otherwise we split the interval $[\tau_j, \tau_{j+1})$ containing d_{n+1} into two intervals $[\tau_j, d_{n+1})$

and $[d_{n+1}, \tau_{j+1})$.

In a next step we add J_{n+1} to the current schedule $S_{OA(m)}$ assuming that the processing volume of J_{n+1} is zero. Let I_j be the interval ending at time d_{n+1}. If $S_{OA(m)}$ has an idle processor in I_j, then we open a new job set $\mathcal{J}_{p+1}$ with associated speed $s_{p+1} = 0$. Otherwise let s_i be the lowest speed used by any of the processors in I_j. We add J_{n+1} to $\mathcal{J}_i$. In the following we increase the processing volume of J_{n+1} from zero to its final size v_{n+1}. The increase proceeds in a series of phases such that, during a phase, the job sets $\mathcal{J}_i$, $i \geq 1$, do not change. However, at the beginning of a phase the job set $\mathcal{J}_{i_0}$ containing J_{n+1} might split into two sets. Moreover, at the end of a phase the set $\mathcal{J}_{i_0}$ might merge with the next set $\mathcal{J}_{i_0-1}$ executed at the next higher speed level. So suppose that during a phase the processing volume of J_{n+1} increases by ϵ. By Lemma 23, the speeds of the jobs in sets $\mathcal{J}_i$, $i > i_0$, do not change. Moreover, by Lemma 22, the speeds of the other jobs can only increase. Hence, by convexity of the power function, an optimal solution will increase the speed at the lowest possible level, which is s_{i_0}. In the current schedule of $OA(m)$, let T_{i_0} be the total time for which jobs of $\mathcal{J}_{i_0}$ are processed. The speed at which jobs of $\mathcal{J}_{i_0}$ are processed increases from $V_{OA(m)}(i_0)/T_{i_0}$ to $(V_{OA(m)}(i_0) + \epsilon)/T_{i_0}$. Hence the potential change is given by

$$\begin{aligned}\Delta\Phi =& \alpha \left(\frac{V_{OA(m)}(i_0) + \epsilon}{T_{i_0}} \right) (V_{OA(m)}(i_0) + \epsilon - \alpha(V_{OPT}(i_0) + \epsilon)) \\ &- \alpha \left(\frac{V_{OA(m)}(i_0)}{T_{i_0}} \right) (V_{OA(m)}(i_0) - \alpha(V_{OPT}(i_0)).\end{aligned}$$

As in [19] we can show the non-positivity of the last expression.

At the beginning of a phase $\mathcal{J}_{i_0}$ might split into two sets $\mathcal{J}'_{i_0}$ and $\mathcal{J}''_{i_0}$. Set $\mathcal{J}'_{i_0}$ contains those jobs that, in an optimal schedule, cannot be executed at an increased speed when raising the processing volume of J_{n+1}. Set $\mathcal{J}''_{i_0}$ contains the jobs for which the speed can be increased. This split into two sets does not change the value of the potential function. For completeness we mention that a job of $\mathcal{J}_{i_0}$ whose execution starts, say, in interval $I_j = [\tau_j, \tau_{j+1})$ belongs to $\mathcal{J}'_{i_0}$ iff in the current schedule (a) no job $J_k \in \mathcal{J}_{i_0}$ with $d_k > \tau_j$ is processed before τ_j or (b) any job $J_k \in \mathcal{J}_{i_0}$ with $d_k > \tau_j$ that is also processed before τ_j is scheduled continuously without interruption throughout $[\tau_j, d_k)$. When the processing volume of J_{n+1} increases as described above, the jobs of $\mathcal{J}'_{i_0}$ remain at their speed level s_{i_0} while the speed of the jobs in $\mathcal{J}''_{i_0}$ increases.

At the end of a phase $\mathcal{J}_{i_0}$ might be merged with the next set $\mathcal{J}_{i_0-1}$. This event occurs if speed value s_{i_0} reaches the next higher level s_{i_0-1}. We always choose ϵ such that $(V_{OA(m)}(i_0) + \epsilon)/T_{i_0}$ does not exceed s_{i_0-1}. Again the merge of two job set does not change the value of the potential function.

Theorem 9. *Algorithm OA(m) is α^α-competitive.*

3.2.2 Algorithm Average Rate

The original *Average Rate (AVR)* algorithm for a single processor considers job densities, where the density δ_i of a job J_i is defined as $\delta_i = v_i/(d_i - r_i)$. At time t the processor speed s_t is set to the sum of the densities of the jobs active at time t. Using these speeds s_t, *AVR* always processes the job having the earliest deadline among the active unfinished jobs.

In the following we describe our algorithm, called *AVR(m)*, for m parallel processors. We assume without loss of generality that all release times and deadlines are integers. Moreover, we assume that the earliest release time $\min_{1\le i\le n} r_i$ is equal to 0. Let $T = \max_{1\le i\le n} d_i$ be the last deadline. For any time interval $I_t = [t, t+1)$ $0 \le t < T$, let $\mathcal{J}_t$ be the set of jobs J_i that are active in I_t, i.e. that satisfy $I_t \subseteq [r_i, d_i)$. Algorithm *AVR(m)* makes scheduling decisions for the intervals I_t over time, $0 \le t < T$. In each I_t, the algorithm schedules δ_i units of v_i, for each job $J_i \in \mathcal{J}_t$. The schedule is computed iteratively. In each iteration there is a set $\mathcal{J}'_t \subseteq \mathcal{J}_t$ of jobs to be scheduled on a subset M of the processors in interval I_t. Initially $\mathcal{J}'_t = \mathcal{J}_t$, and M contains all the processors of the system. Furthermore, let $\Delta_t = \sum_{J_i\in\mathcal{J}_t} \delta_i$ be the total density of jobs active at time t, and $\Delta'_t = \sum_{J_i\in\mathcal{J}'_t} \delta_i$ be the total density of the jobs in $\mathcal{J}'_t$. The average load of the jobs in $\mathcal{J}_t$ and $\mathcal{J}'_t$ is Δ_t/m and $\Delta'_t/|M|$, respectively.

In each iteration, if $\max_{J_i\in\mathcal{J}'_t} \delta_i > \Delta'_t/|M|$, then a job J_i of maximum density among the jobs in $\mathcal{J}'_t$ is scheduled on a processor $l \in M$. Job J_i is processed during the whole interval I_t at a speed of $s_{t,l} = \delta_i$. Then $\mathcal{J}'_t$ and M are updated to $\mathcal{J}'_t \setminus \{J_i\}$ and $M \setminus \{l\}$, respectively. On the other hand if $\max_{J_i\in\mathcal{J}'_t} \delta_i \le \Delta'_t/|M|$, the complete set $\mathcal{J}'_t$ is scheduled at a uniform speed of $s_\Delta = \Delta'_t/|M|$ on the processors of M as follows. First a sequential schedule S of length $|M|$ is constructed by concatenating the execution intervals of all jobs in $\mathcal{J}'_t$. Again, each job in $\mathcal{J}'_t$ is executed at a speed of s_Δ and therefore has an execution interval of length δ_i/s_Δ. Then S is split among the processors of M by assigning time range $[(\mu-1), \mu)$ of S from left to right to the μ-th processor of M, $1 \le \mu \le |M|$. A summary of *AVR(m)* is given in Figure 3.3.

It is easy to see that *AVR(m)* computes feasible schedules. In any interval I_t where a job J_i is active, δ_i processing units of J_i are finished. Summing over all intervals where J_i is active, we obtain that exactly $\delta_i(d_i - r_i) = v_i$ processing units, i.e. the total work of J_i, are completed. Furthermore, at no point in time is a job scheduled simultaneously on two different processors: In any interval I_t, a job $J_i \in \mathcal{J}_t$ is either scheduled alone on just one processor or it is scheduled at a speed of s_Δ. In the second case, since J_i has a density of $\delta_i \le s_\Delta$ its execution interval in S has a length of at most one time unit. Therefore, even if J_i is split among two processors μ and $\mu+1$, $1 \le \mu < |M|$, it is processed at the end of processor μ and at the beginning of $\mu+1$ so that these two execution intervals do not overlap (recall that $|I_t| = 1$).

```
Algorithm AVR(m): In each interval $I_t$, $0 \le t < T$, execute the following.

1. $\mathcal{J}'_t := \mathcal{J}_t$; $M$ is the set of all processors in the system;
2. while $\mathcal{J}'_t \neq \emptyset$ do
3.     while $\max_{j_i \in \mathcal{J}'_t} \delta_i > \Delta'_t/|M|$ do
4.         Pick a processor $l \in M$; Schedule $J_i := \arg\max_{J_i \in \mathcal{J}'_t} \delta_i$ at a
             speed of $\delta_i$ on processor $l$;
5.         $\mathcal{J}'_t := \mathcal{J}'_t \setminus \{J_i\}$; $M := M \setminus \{l\}$;
6.     $s_\Delta := \Delta'_t/|M|$; Feasibly schedule every job $J_i \in \mathcal{J}'_t$ at a speed of $s_\Delta$.
```

Figure 3.3: The algorithm *AVR(m)*.

Theorem 10. *AVR(m) achieves a competitive ratio of* $(2\alpha)^\alpha/2 + 1$.

Proof. Let $\sigma = J_1, \ldots, J_n$ be an arbitrary job sequence. The energy incurred by $AVR(m)$ is

$$E_{AVR(m)}(\sigma) = \sum_{t=0}^{T-1} \sum_{l=1}^{m} (s_{t,l})^\alpha |I_t|.$$

The length $|I_t|$ of each interval I_t is equal to 1 time unit, but we need to incorporate $|I_t|$ in the above expression in order to properly evaluate the consumed energy.

For each I_t, we partition the m processors into two sets depending on whether or not the actual speeds are higher than Δ_t/m. More precisely, let $M_{t,1}$ be the set of all l, $1 \le l \le m$, such that the speed of processor l is $s_{t,l} \le \frac{\Delta_t}{m}$. Let $M_{t,2} = \{1, \ldots, m\} \setminus M_{t,1}$. We have

$$\begin{aligned} E_{AVR(m)}(\sigma) &\le \sum_{t=0}^{T-1} \sum_{l \in M_{t,1}} \left(\frac{\Delta_t}{m}\right)^\alpha |I_t| + \sum_{t=0}^{T-1} \sum_{l \in M_{t,2}} (s_{t,l})^\alpha |I_t| \\ &\le \sum_{t=0}^{T-1} m^{1-\alpha} (\Delta_t)^\alpha |I_t| + \sum_{t=0}^{T-1} \sum_{l \in M_{t,2}} (s_{t,l})^\alpha |I_t|. \end{aligned}$$

The last inequality holds because $|M_{t,1}| \le m$. Next, for any fixed I_t, consider the processors of $M_{t,2}$. By the definition of $AVR(m)$ any such processor is assigned exactly one job. Hence, for any processor l with $l \in M_{t,2}$, the speed $s_{t,l}$ is the density of the job assigned to it. Hence $\sum_{l \in M_{t,2}} (s_{t,l})^\alpha \le \sum_{J_i \in \mathcal{J}_t} (\delta_i)^\alpha$ and we obtain

$$E_{AVR(m)}(\sigma) \le m^{1-\alpha} \sum_{t=0}^{T-1} (\Delta_t)^\alpha |I_t| + \sum_{t=0}^{T-1} \sum_{J_i \in \mathcal{J}_t} (\delta_i)^\alpha |I_t|. \tag{3.9}$$

In the above expression $\sum_{t=0}^{T-1}\sum_{J_i\in\mathcal{J}_t}(\delta_i)^\alpha|I_t|$ each job contributes an energy of $(\delta_i)^\alpha(d_i - r_i)$ because $J_i \in \mathcal{J}_t$ for any $t \in [r_i, d_i)$. This amount is the minimum energy required to process J_i if no other jobs were present. Hence $\sum_{t=0}^{T-1}\sum_{J_i\in\mathcal{J}_t}(\delta_i)^\alpha|I_t| \le E_{OPT}(\sigma)$, where $E_{OPT}(\sigma)$ is the energy consumption of an optimal schedule for σ on the m processors.

As for the first term in (3.9), $\sum_{t=0}^{T-1}(\Delta_t)^\alpha|I_t|$ is the energy consumed by the single processor algorithm *AVR* when executing σ on one processor. Recall that *AVR* sets the speed at time t to the accumulated density of jobs active at time t. Since *AVR* achieves a competitive ratio of $(2\alpha)^\alpha/2$, see [55], we obtain $\sum_{t=0}^{T-1}(\Delta_t)^\alpha|I_t| \le \frac{1}{2}(2\alpha)^\alpha E^1_{OPT}(\sigma)$, where $E^1_{OPT}(\sigma)$ is the energy of an optimal single processor schedule for σ. Thus

$$E_{AVR(m)}(\sigma) \le m^{1-\alpha}\tfrac{1}{2}(2\alpha)^\alpha E^1_{OPT}(\sigma) + E_{OPT}(\sigma).$$

To establish the desired competitive ratio, we finally prove $m^{1-\alpha}E^1_{OPT}(\sigma) \le E_{OPT}(\sigma)$.

Consider an optimal schedule S_{OPT} for σ on m processors and let $s^o_{t,l}$ be the speed on processor l in I_t, where $1 \le l \le m$ and $0 \le t < T-1$. Let $s^o_t = \sum_{l=1}^m s^o_{t,l}$ be the sum of the speeds in I_t. By the convexity of the power function

$$E_{OPT}(\sigma) = \sum_{t=0}^{T-1}\sum_{l=1}^{m}(s^o_{t,l})^\alpha|I_t| \ge \sum_{t=0}^{T-1} m(s^o_t/m)^\alpha|I_t| = m^{1-\alpha}\sum_{t=0}^{T-1}(s^o_t)^\alpha|I_t|. \tag{3.10}$$

Finally, consider the single processor schedule S^1 that, in any interval I_t, uses speed s^o_t and accomplishes the same work as S_{OPT} in I_t. More precisely, if S_{OPT} finishes $v_{t,i}$ processing units of J_i in I_t, where $0 \le v_{t,i} \le v_i$, then S^1 processes the same amount of J_i in I_t. Using speed s^o_t, this is indeed possible. Hence S^1 is a feasible schedule whose energy consumption is $\sum_{t=0}^{T-1}(s^o_t)^\alpha|I_t|$, and this expression is at least $E^1_{OPT}(\sigma)$. Combining with (3.10) we obtain

$$E_{OPT}(\sigma) \ge m^{1-\alpha}\sum_{t=0}^{T-1}(s^o_t)^\alpha|I_t| \ge m^{1-\alpha}E^1_{OPT}(\sigma). \qquad \square$$

Chapter 4

Load Balancing with Unit-Weight Jobs

When scheduling in a multi-processor environment, it is often desirable to keep the load of the jobs to be processed as balanced as possible across the processors. A higher load than necessary on a processor can for instance result in worse performance than what is actually possible. Furthermore a higher load on a processor can cause a higher working temperature, leading to hardware damage in the long run.

We consider the following load balancing problem: We are given a set $\mathcal{I} = \{I_1, \dots, I_n\}$ of jobs, where each job is represented by an interval $I = [\ell, r) \in \mathcal{I}$ with starttime ℓ and endtime r. Furthermore, we are given k processors and have to assign the jobs to the processors as evenly as possible. That is, we want to minimize the maximal difference of the numbers of jobs processed by any two processors over all times.

As an example, suppose that we are given k production sites and have to carry out a set $\mathcal{I}$ of requests. For each request $I = [\ell, r) \in \mathcal{I}$, let ℓ be the time when production has to begin and let $r - \ell$ be the time required to produce the respective good. Now, the goal is to distribute the requests to the k sites as evenly as possible in the above sense.

More formally, our problem is that of scheduling unit-weight jobs, on identical machines, with the objective of current load. As we have already seen, we can reformulate this load balancing problem in terms of the following more intuitive interval coloring problem. We are given a set $\mathcal{I} = \{I_1, \dots, I_n\}$ of n intervals on the real line and a set $K = \{1, \dots, k\}$ of k colors. A *k-coloring* is a mapping $\chi : \mathcal{I} \to K$. For a fixed k-coloring χ and a point $x \in \mathbb{R}$, let $c_i(x)$ denote the number of intervals containing x that have color i in χ. Define the *imbalance* of

χ at x by

$$\operatorname{imb}(x) = \max_{i,j \in K} |c_i(x) - c_j(x)|. \tag{4.1}$$

In words, this is the maximum difference in the size of color classes at point x. The *imbalance* of χ is given by $\operatorname{imb}(\chi) = \max_{x \in \mathbb{R}} \operatorname{imb}(x)$.

These definitions yield the following minimization problem:

MINIMUM IMBALANCE INTERVAL k-COLORING
Instance: A set of intervals $\mathcal{I}$.
Task: Find a k-coloring χ with minimal $\operatorname{imb}(\chi)$.

We call a k-coloring with imbalance at most one *balanced*. Observe that if an odd number of intervals intersect at some point, imbalance at least one is unavoidable. Moreover, if a balanced coloring exists, its imbalance is obviously minimal. As we will see, it can be deduced from known results that it is always possible to find a balanced interval k-coloring. Hence in this chapter we will mostly focus on the construction of balanced interval k-colorings. We also study several extensions and generalizations of the problem.

Previous Work

We focus our review on machine load balancing to problems considering temporary jobs, that is, endtimes of jobs can take any value. In [12], Azar et al. study a problem with arbitrary weighted jobs that is similar to the one we consider with unit-weight jobs. Both settings focus on the offline setting and identical machines. The difference lies in the objective function: they consider minimizing the peak load, whereas we minimize the current load. For their problem they give a polynomial time approximation scheme when the number of machines is fixed and show that no algorithm can achieve an approximation ratio better than $3/2$ unless P=NP when the number of machines is part of the input. Armon et al. [9] study the problem on unrelated machines, and show again a polynomial time approximation scheme for a fixed number of machines. Furthermore they prove an innaproximability factor of two for the case when the number of machines is part of the input.

A variation of our problem with arbitrary weighted jobs was studied by Westbrook [54]. However, in [54] it is allowed to reassign a job to a different machine (at some cost). Westbrook considered the current load objective and has developed competitive algorithms in both the identical and related machine settings. Finally, Andrews, Goemans and Zhang in [8] and subsequently Skutella and Verschae in [49] have improved upon several of these results. For a survey on online load balancing, see [11].

MINIMUM IMBALANCE INTERVAL k-COLORING also has close connections to discrepancy theory; see Doerr [28] and Matoušek [47] for introductions to the field. Let $H = (X, U)$ be a hypergraph consisting of a set X of vertices and a set $U \subseteq 2^X$ of hyperedges. Analogous to the previous definitions, a k-coloring is a mapping $\chi : X \to K$, and the imbalance $\operatorname{imb}(\chi)$ is the largest difference in size between two color classes over all hyperedges. The *discrepancy* problem is to determine the smallest possible imbalance, i. e., $\operatorname{disc}(H) = \min_{\chi:X\to K} \operatorname{imb}(\chi)$.

Our problem can be reformulated to that of finding the discrepancy of the hypergraph $H = (X, U)$, where the intervals $\mathcal{I}$ are identified with X, and U is the family of all maximal subsets of intervals intersecting at some point. It can be shown that this hypergraph has a totally unimodular incidence matrix. This is useful because de Werra [53] proved that balanced k-colorings exist for hypergraphs with totally unimodular incidence matrices. However, the proof in [53] is only partially constructive: a balanced k-coloring is constructed by iteratively solving the problem of balanced 2-coloring on hypergraphs with a totally unimodular incidence matrix, for which only existence is known.

Further related work in discrepancy theory mostly considers hypergraph coloring with two colors and often from existential, rather than algorithmic perspective. For an arbitrary hypergraph H with n vertices and m hyperedges, the bound $\operatorname{disc}(H) \le \sqrt{2n \ln(2m)}$ for 2-coloring follows with the probabilistic method; see also [28]. For $m \ge n$, Spencer [51] proved the stronger result $\operatorname{disc}(H) = O(\sqrt{n \log(m/n)})$, which is in particular interesting for $m = O(n)$. If each vertex is contained in at most t edges, the 2-coloring bound $\operatorname{disc}(H) = O(\sqrt{t} \log n)$ was shown by Srinivasan [52] and the bound $\operatorname{disc}(H) \le 2t - 1$ by Beck and Fiala [22]. Biedl et al. [23] improved the bound to $\operatorname{disc}(H) \le \max\{2t-3, 2\}$ for 2-colorings and established $\operatorname{disc}(H) \le 4t - 3$ for general k-colorings. They also showed that it is NP-complete to decide the existence of balanced k-colorings for hypergraphs with $t \ge \max\{3, k-1\}$ and $k \ge 2$.

Bansal [14] recently gave efficient algorithms that achieve 2-color imbalances similar to [51,52] up to constant factors. In particular, an algorithm yields $\operatorname{disc}(H) = O(\sqrt{n} \log(2m/n))$ matching the result of Spencer [51] if $m = O(n)$. Furthermore, $\operatorname{disc}(H) = O(\sqrt{t} \log n)$ complies with the non-constructive result of Srinivasan [52]. For general $k > 2$, Doerr and Srivastav [29] gave a recursive method constructing k-colorings from (approximative) 2-colorings.

Unfortunately, these results on general discrepancy theory are not helpful for the considered problem, because t is only bounded by the number of vertices.

Our Contribution

We investigate several questions related to MINIMUM IMBALANCE INTERVAL k-COLORING, and extensions thereof in Section 4.1. As we have already mentioned, balanced k-colorings exist for any set $\mathcal{I}$ of intervals. We establish this in Subsection 4.1.1 by showing that our hypergraph H has totally unimodular incidence matrix and then applying a result of de Werra [53]. It also follows independently from our algorithmic results below.

Additionally, we show that a balanced k-coloring can be constructed in polynomial time. We first present an $O(n \log n)$ algorithm for finding a balanced 2-coloring in Subsection 4.1.2, thereby establishing the first constructive result for intervals. Recall that de Werra [53] did not give a construction for a balanced 2-coloring. Then, in Subsection 4.1.3, we give an $O(n \log n + kn \log k)$ algorithm for finding a balanced k-coloring. This is an improvement in time complexity, since the construction of de Werra [53] combined with our algorithm for 2-coloring only yields $O(n \log n + k^2 n)$. We also note that our algorithm works for any hypergraph with incidence matrix having the consecutive-ones property.

An interesting extension of the problem, is to consider arcs of a circle instead of intervals. This setting can, for example, model periodic jobs. For arcs of a circle, balanced k-colorings do not exist in general. However, in Subsection 4.1.4 we give an algorithm achieving imbalance at most two with the same time complexity as in the interval case.

The online scenario, in which we learn intervals over time, is also interesting. We show in Subsection 4.1.5 that the imbalance of *any* online algorithm can be made arbitrarily high, and therefore a competitive online algorithm for the problem is not possible.

Finally, we consider the generalization of the problem to d-dimensional boxes (instead of intervals), and to multiple intervals. In Section 4.2, we show that it is NP-complete to decide if a balanced k-coloring exists for d-dimensional boxes, for any $d \geq 2$ and any $k \geq 2$. Our reduction is from NOT-ALL-EQUAL 3SAT. This result clearly implies NP-hardness of the respective minimization problem. Additionally, we show the NP-hardness of the more general problem where every job can have multiple disjoint active intervals.

4.1 Interval Colorings

In this section, we consider MINIMUM IMBALANCE INTERVAL k-COLORING, establish the existence of balanced k-colorings, and give algorithms for 2 and k colors, respectively. Later, we consider arcs of a circle and an online version.

4.1.1 Existence of Balanced k-Colorings

We begin by observing the existence of balanced k-colorings. In the proof below, we use a theorem of de Werra [53], but the existence of balanced k-colorings also follows from our algorithmic results.

Theorem 11. *For any set $\mathcal{I}$ of intervals and any $k \in \mathbb{N}$, there is a balanced k-coloring.*

Proof. Let $\mathcal{I}$ be the set of given intervals. Define a hypergraph $H = (X, U)$ by identifying X and $\mathcal{I}$ and by defining U to be the family of all maximal subsets of intervals intersecting at some point. For H with $X = \{x_1, \dots, x_n\}$ and $U = \{U_1, \dots, U_m\}$, the incidence matrix is defined by $A = (a_{i,j})$ with $a_{i,j} = 1$ if $x_i \in U_j$ and $a_{i,j} = 0$ otherwise.

De Werra [53] showed that any hypergraph with totally unimodular incidence matrix admits a balanced k-coloring. It is well-known that a 0–1-matrix is totally unimodular if it has the consecutive-ones property, i. e., if there is a permutation of its columns such that all 1-entries appear consecutively in every row. The incidence matrix A of H has this property:

We can just order the U_j in increasing order of intersection, in order to have the entries $a_{i,j} = 1$ appearing consecutively in each row. □

4.1.2 Algorithm for Two Colors

In this subsection, we present an algorithm that constructs a balanced 2-coloring in polynomial time. Since the algorithm produces a valid solution for every possible instance, Theorem 11 for $k = 2$ also follows from this algorithmic result. The main idea is to simplify the structure of the instance such that the remaining intervals have start- and endpoints occurring pairwise. We then build a constraint graph that has the intervals as vertices. Finally, a proper 2-coloring of the constraint graph induces a solution to the problem.

We refer to the start- and endpoints of the intervals as *events*. Additionally, we call intervals spanned by two consecutive events as *regions*. A region is called even (odd) if an even (odd) number of input intervals are active in this region. Without loss of generality, we can assume that the start- and endpoints of the input intervals are pairwise disjoint. If not, a new instance can be obtained by repeatedly increasing one of the coinciding start- or endpoints by $\epsilon/2$, where ϵ is the minimum size of a region. Since the new instance includes a region for every region of the original instance, a balanced coloring for the new instance implies a balanced coloring for the old instance (the converse is not true).

Theorem 12. *For any set $\mathcal{I}$ of n intervals, there is a balanced 2-coloring that can be constructed in $O(n \log n)$ time.*

Proof. Observe that a coloring that is balanced on all even regions is also balanced on all odd regions. This is because odd regions only differ by one interval from a neighboring even region. Thus, it suffices to construct a balanced coloring of the even regions. Since between two consecutive even regions, exactly two events occur, it suffices to consider only pairs of consecutive events.

If a pair of events consists of the start- and endpoint of the same interval, this interval is assigned any color and is removed from the instance. If a pair consists of start- and endpoint of different intervals, these intervals are removed from the instance and substituted by a new interval (of minimal length) that covers their union. In a final step of the algorithm, both intervals will be assigned the color of their substitution. The remaining instance consists solely of pairs of events where two intervals start or two intervals end. Clearly, a balanced coloring has to assign opposite colors to the corresponding two intervals of such a pair, and any such assignment yields a balanced coloring.

The remaining pairs of events induce a *constraint graph*. Every vertex corresponds to an interval, and if two intervals start or end pairwise, then an edge is added between them. Finding a proper vertex two-coloring of this graph gives a balanced 2-coloring. The constraint graph is bipartite: Each edge can be labeled by "$\vdash$" or "$\dashv$" if it corresponds to two start- or endpoints, respectively. Since each interval is incident to exactly two edges, any path must traverse $\vdash$- and $\dashv$-edges alternatingly. Therefore, every cycle must be of even length and hence the graph is bipartite, and a proper vertex two-coloring of the constraint graph can be found in linear time by depth-first search.

We denote that sorting events takes $O(n \log n)$ time, and the creation of the constraint graph and its coloring takes linear time. □

Note that if intervals are given already sorted, or interval endpoints are described by small integers, then the above algorithm can even find a balanced 2-coloring in linear time.

4.1.3 Algorithms for k Colors

In this subsection, we extend the results of the previous subsection to an arbitrary number of colors k and show that a balanced interval k-coloring can be found in polynomial time.

A first polynomial time algorithm can be obtained using a construction by de Werra [53]: Start with an arbitrary coloring and find two colors i and j for which $\max_x |c_i(x) - c_j(x)|$ is maximal. Use the algorithm from Subsection 4.1.2 to find a balanced 2-coloring of all intervals that currently have color i or j and recolor them accordingly. Repeat until the coloring is balanced. This algorithm has running time $O(n \log n + k^2 n)$, because sorting intervals is needed only once for the above algorithm for 2 colors, and there are at most $\binom{k}{2}$ recolorings necessary.

In the following, we present an alternative algorithm for k colors, which is faster than $O(n \log n + k^2 n)$. We will first give an overview of the argument, and then a more formal description.

As in Subection 4.1.2, we assume without loss of generality that all start- and endpoints are pairwise disjoint. The idea is to scan the events in order, beginning with the first in time, and to capture dependencies in k-tuples of intervals that indicate pairwise different colors. That is, we reduce the MINIMUM IMBALANCE INTERVAL k-COLORING instance to an instance of STRONG HYPERGRAPH COLORING, formally defined as follows.

STRONG HYPERGRAPH COLORING

Instance: A ground set X, a family $\mathcal{S}$ of constraints $S_1, \ldots, S_n \subseteq X$, and an integer k.
Task: Find a k-coloring $\chi : X \to K$ with $\forall\, 1 \leq i \leq n : x, y \in S_i, x \neq y \Rightarrow \chi(x) \neq \chi(y)$.

For example, in the special case that each block of k consecutive events consists only of start- or only of endpoints, the constraints that the corresponding k intervals have to be differently colored will capture the whole solution. As we will see below, also different interval nesting structures can be captured by such constraints.

STRONG HYPERGRAPH COLORING is NP-hard in general [4]. However, each interval will occur in at most two constraints, corresponding to its start- and endpoint. Thus, we can further reduce the special STRONG HYPERGRAPH COLORING instances to EDGE COLORING instances, where the goal is to color edges of a multigraph such that the edges incident to each vertex are all differently colored. More formally:

EDGE COLORING

Instance: A multigraph $G = (V, E)$ and an integer k.
Task: Find a k-coloring $\chi : E \to K$ with $\forall e, e' \in E, e \cap e' \neq \emptyset, e \neq e' \Rightarrow \chi(e) \neq \chi(e')$.

It is easy to see that any instance $(X, \mathcal{S}, k)$ of STRONG HYPERGRAPH COLORING where each element of X occurs in at most two constraints can be reduced to EDGE COLORING as $(G = (\mathcal{S}, E), k)$, where for each $x \in X$ that occurs in S_i and S_j with $i \neq j$, we add the edge $\{S_i, S_j\}$ to E. For our case, a constraint corresponds to a vertex, and an interval corresponds to an edge that connects the

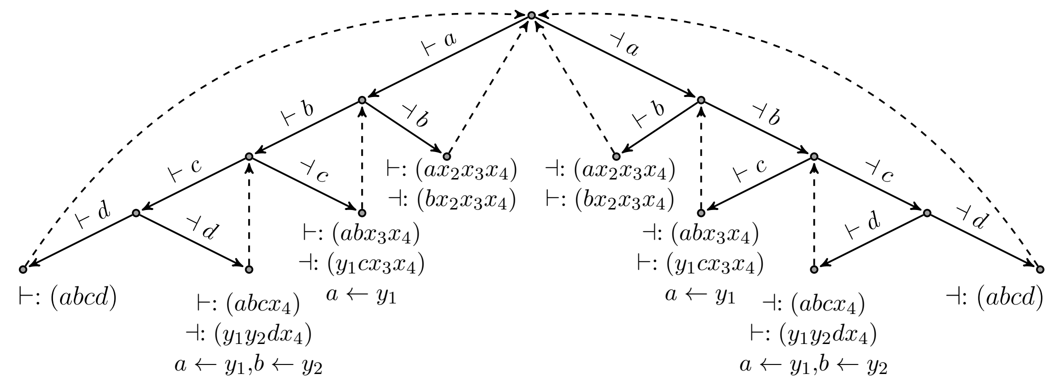


Figure 4.1: Tracking of active events for $k = 4$

two constraints it occurs in. An edge coloring with k colors will thus provide a balanced interval k-coloring.

Clearly, an edge coloring of a multigraph with maximum degree Δ needs at least Δ colors. Finding an edge coloring of minimum size is NP-hard in general [37]. However, Kőnig [42] showed that for bipartite multigraphs, Δ colors always suffice. Further, an edge coloring of a bipartite multigraph with m edges can be found in $O(m \log \Delta)$ time [27]. The multigraph that we construct has maximum degree k and is bipartite. Thus, a balanced interval k-coloring always exists and can be found in polynomial time.

We now describe the construction of the constraints and give a more rigorous description and formal proofs of the results. From a MINIMUM IMBALANCE INTERVAL k-COLORING instance $\mathcal{I}$, we construct a STRONG HYPERGRAPH COLORING instance $(\mathcal{I}, \mathcal{S}, k)$ over the ground set of the intervals. The algorithm scans the set of events in order, beginning with the smallest. It keeps a set of active events and adds constraints to $\mathcal{S}$. Initially, the set of active events is empty. Then events are added as they appear over time, and every time a region is reached where the number of intervals is 0 modulo k, the set of active intervals gets reset to empty. Thus, the active events always describe the change from a situation where each color occurs the same number of times.

The construction of the constraints can be visualized with a decision tree, depicted for the example $k = 4$ in Figure 4.1. At the beginning, the set of active events is empty (which corresponds to the root of the decision tree). Whenever the set of events is empty, the algorithm branches into two cases depending on the type of the next event. Both branches are equivalent, with the roles of start- and endpoints interchanged. Therefore, assume the next event is the start of an interval (depicted as $\vdash a$ on the left branch). It is added to the active set. This continues until either k startpoints of intervals $I_1, \ldots, I_k$ are added, or an endpoint is encountered. In the first case, the constraint

$$(I_1, \dots, I_k) \tag{4.2}$$

is constructed and the set of active events is reset (dashed arrow returning to the root). In the second case, assume the startpoints of intervals $I_1, \dots, I_j$ have been added and the endpoint of interval I_{j+1} is encountered. Then, the two constraints

$$(I_1, \dots, I_j, x_{j+1}, \dots, x_k) \tag{4.3}$$
$$(y_1, \dots, y_{j-1}, I_{j+1}, x_{j+1}, \dots, x_k) \tag{4.4}$$

are constructed. Here, $x_{j+1}, \dots, x_k$ and $y_1, \dots, y_{j-1}$ are new axiliary intervals that have not been used in previous constraints. That is, they do not correspond to actual intervals of the real line and only serve as placeholders in the active set. Furthermore, the startpoints of the intervals $I_1, \dots, I_j$ are replaced by the startpoints of $y_1, \dots, y_{j-1}$ in the active set of events (indicated by the dashed arrows pointing one level higher in the decision tree).

We now prove the correctness of the two chained reductions.

Lemma 24. *A solution to the* STRONG HYPERGRAPH COLORING *instance* $(\mathcal{I}, \mathcal{S}, k)$*, constructed as described above, yields a balanced* k*-coloring for* $\mathcal{I}$*.*

Proof. Recall that a region is an interval spanned by two consecutive events. The proof is by induction over all regions, in the order of events. At each region, we count the number of times each of the k colors is used among the intervals containing the region. We will show that these counters differ by at most one, i. e., the coloring is balanced. Clearly, all counters are equal to zero before the first event.

In particular, we show that in regions where the number of intervals is 0 modulo k, all counters are equal, and that between these regions the counters change by at most one and all in the same direction. We distinguish the same cases as in the construction. We consider that the first event is a startpoint. The case where the first event encountered is an endpoint, follows by a symmetrical argument.

If k startpoints of intervals $I_1, \dots, I_k$ are encountered, there is a constraint of the form (4.2), which ensures that all of them have different colors. Therefore, at each startpoint, a different counter increases by one. Hence, the counters differ by at most one in all regions up to the startpoint of I_k, and are all equal in the region beginning with the startpoint of I_k.

Consider that only $j < k$ startpoints of the intervals $I_1, \dots, I_j$ are encountered. Since these startpoints were added to the set of active events during the construction, the intervals $I_1, \dots, I_j$ do all occur in one constraint. This constraint forces them to have different colors, and therefore the colors of $I_1, \dots, I_j$ are exactly the colors with increased count. Now, we distinguish two subcases depending on the next event. If the next event is a startpoint of some interval, denoted by I_{j+1}, it will also be part of the same constraint. Hence, I_{j+1} has a different color whose count is not yet increased. The second subcase is that the next event is the endpoint of some interval, denoted by I_{j+1}. The interval I_{j+1} is forced to have the same color as one of the intervals $I_1, \dots, I_j$. This is because there is a constraint of the form (4.3) that collects all other colors in variables $x_{j+1}, \dots, x_k$, which occur together with I_{j+1} in a constraint of the form (4.4). Thus, the previously increased counter for the color of I_{j+1} decreases again. Because of the constraint of the form (4.3), the virtual intervals $y_1, \dots, y_{j-1}$ must have all of the colors of $I_1, \dots, I_j$ except for the color of I_{j+1}. Hence, the colors of $y_1, \dots, y_{j-1}$ are exactly all remaining colors with increased count. Therefore, we are in the same situation as before encountering the endpoint of I_{j+1}. By repeating the above argument, the claim follows also in this case. □

Lemma 25. *A* STRONG HYPERGRAPH COLORING *instance* $(\mathcal{I}, \mathcal{S}, k)$ *constructed as described above can be reduced to a bipartite* EDGE COLORING *instance.*

Proof. We need to show that each interval occurs in at most two constraints. Once this is proved, it is possible to build a multigraph with the constraints as vertices and edges between them if they share a common interval. Further, it has to be shown that this multigraph is bipartite. To show both parts at once, we color the constraints in $\mathcal{S}$ with the two colors $\vdash$ and $\dashv$. It then suffices to show that every interval can occur in at most one $\vdash$-constraint and in at most one $\dashv$-constraint.

We color a constraint with $\vdash$ when the involved nonvirtual intervals startpoint is removed from the active set, and with $\dashv$ otherwise (see Figure 4.1). All nonvirtual intervals therefore occur in exactly two constraints, constructed when the start- and endpoint get removed from the active set of events. A virtual x-interval always occurs in a pair of subsequent differently colored constraints, and is not used anywhere else. For the left branch of the decision tree, a virtual y-interval occurs first in a $\dashv$-constraint, and its startpoint is then added to the list of active events. Then, it will be used in a constraint of type either (4.2) or (4.3), both of which are of type $\vdash$. The argument is symmetrical for the right branch of the decision tree. □

Theorem 13. *Every set of n intervals $\mathcal{I}$ has a balanced k-coloring for any $k \in \mathbb{N}$, and it can be found in $O(n \log n + kn \log k)$ time.*

Proof. By Lemmas 24 and 25, MINIMUM IMBALANCE INTERVAL k-COLORING can be reduced to EDGE COLORING with fixed k in a bipartite multigraph. The maximum degree of this multigraph is k, since by construction no constraint has more than k elements. By Kőnig's theorem [42] existence follows.

To be able to process the events, they have to be sorted in $O(n \log n)$ time. We have $O(kn)$ virtual intervals and thus $O(kn)$ edges in the EDGE COLORING instance. Finding an edge k-coloring for this multigraph with maximum degree k can be done in $O(kn \log k)$ time [27]. □

Note that the EDGE COLORING algorithm by Cole, Ost, and Schirra [27] uses quite involved data structures. In practice, it might be preferable to use the much simpler algorithm by Alon [7] running in $O(m \log m)$ time for an m-edge graph, which gives a worst-case bound of $O(kn \log n)$.

Our result can be generalized to arbitrary hypergraphs with the consecutive ones property. Recall that a matrix has the consecutive-ones property if there is a permutation of its columns such that all 1-entries appear consecutively in every row. Such a permutation can be found in linear time by the PQ-algorithm [25]. Given such a matrix, it is straightforward to construct an instance of MINIMUM IMBALANCE INTERVAL k-COLORING.

Theorem 14. *For any hypergraph H with an $n \times m$ incidence matrix having the consecutive ones property, a balanced k-coloring can be found in $O(nm + kn \log k)$ time.*

4.1.4 Arcs of a Circle

In a periodic setting, the jobs $\mathcal{I}$ might be better described by a set of arcs of a circle rather than a set of intervals. In this case, there are instances that require an imbalance of two (e.g., three arcs that intersect exactly pairwise). We show that two is also an upper bound and a coloring with maximal imbalance two can be found in polynomial time.

Theorem 15. *The maximal imbalance for arcs of a circle is two, and finding a coloring with imbalance at most two can be done in $O(n \log n + kn \log k)$ time.*

Proof. Define a point on the circle, called zero, and consider counterclockwise orientation. We build an instance of MINIMUM IMBALANCE INTERVAL k-COLORING by "unfolding" the circle at zero in the following way. Consider only arcs that do not span the full circle. Map all such arcs not containing zero to intervals of the same length at the same distance from zero on the real line. Map the arcs containing zero to intervals of same length such that the positive part of the interval has the same length as the part of the arc to the right of zero. Finally, map the arcs containing the full circle to intervals spanning all of the instance constructed so far. Use the above algorithm to obtain a coloring of the intervals with imbalance at most one at every point. By reversing the mapping, the obtained coloring of the arcs has imbalance at most two (each point on the circle is mapped to at most two points of the real line). □

4.1.5 Online Algorithms

Most of the machine load balancing literature focuses on the online scenario, where not all information is known in advance. In our setting, this means that intervals arrive in order of their starttimes, including the information of their endtimes, and a color has to be assigned to them immediately.

The problem of finding a proper coloring (i. e., a coloring where no two intersecting intervals have the same color) of intervals in an online setting has found considerable interest [3,30,40]. In these works, the objective is to use a minimum number of colors. In contrast, we consider a fixed number of colors and our objective is that of minimizing the imbalance. We show that in contrast to the offline scenario, here the imbalance can become arbitrarily large.

Theorem 16. *In online* MINIMUM IMBALANCE INTERVAL k-COLORING, *the imbalance is unbounded.*

Proof. We first consider the case $k = 2$ with colors "$+1$" and "-1". Denote the signed imbalance $\operatorname{simb}(x)$ to be the sum of the colors of the intervals containing x. Note that $\operatorname{imb}(x) = |\operatorname{simb}(x)|$.

In the following, we outline how an adversary can construct a sequence of intervals such that no online algorithm can yield a bounded imbalance. Initially, $\operatorname{simb} = 0$. Set $L = [0, 1)$ and $R = [2, 3)$, and let L_ℓ, R_ℓ, L_r, and R_r denote the start- and endpoints of the current L and R, respectively. Repeat the following steps.

- Present the interval $[(L_\ell + L_r)/2, (R_\ell + R_r)/2)$ to the online algorithm.
- If it chooses color $+1$, set $R \leftarrow [R_\ell, (R_\ell + R_r)/2)$, else $R \leftarrow [(R_\ell + R_r)/2, R_r)$.
- Set $L \leftarrow [(L_\ell + L_r)/2, L_r)$.

A small example of such a sequence of intervals is depicted in Figure 4.2.

This sequence of intervals is legal, since the startpoints increase strictly monotonously. In each repetition, if the algorithm chooses color $+1$, the signed imbalances in L and R increase by one. If the algorithm chooses -1, the signed imbalance decreases by one in L and remains unchanged in R, i. e., the difference of the signed imbalance in L and R increases. Therefore, the signed imbalance diverges in L or R. Since the imbalance is the absolute value of the signed imbalance, it becomes unbounded.

The construction easily generalizes to $k > 2$ colors. We only track two arbitrary colors, and whenever the algorithm assigns an untracked color to an interval, we present the same interval (with a slightly increased startpoint) again, forcing it to eventually assign a tracked color or to produce unbounded imbalance. □

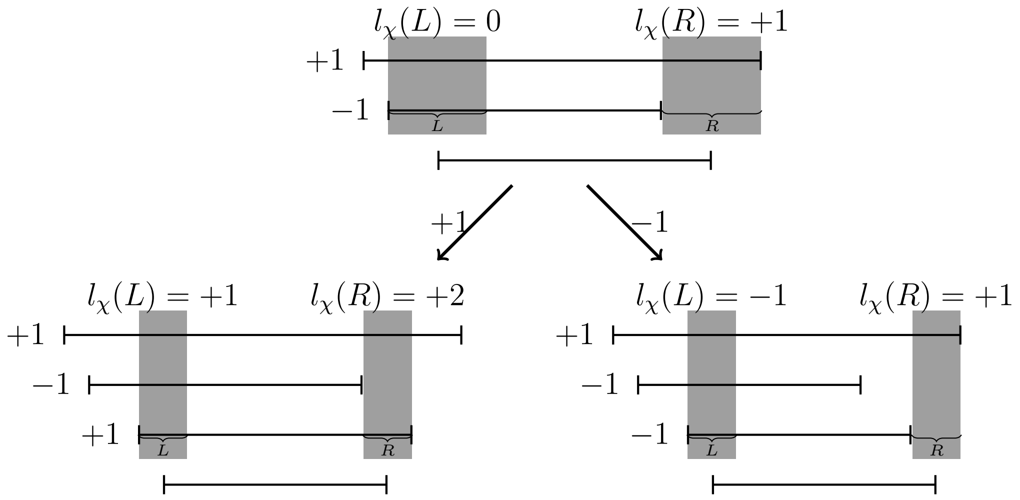


Figure 4.2: A small example of intervals.

4.2 Hardness of Generalizations

We consider generalizations of MINIMUM IMBALANCE INTERVAL k-COLORING and show that they are NP-hard. Note that the hardness results of Biedl et al. [23] do not apply to the problems we consider here.

4.2.1 d-Dimensional Boxes.

Gyárfás and Lehel [35] suggest to examine d-dimensional boxes as generalizations of intervals for coloring problems. The problem MINIMUM IMBALANCE d-BOX k-COLORING has as input an integer k and a set $\mathcal{I} = \{I_1, \ldots, I_n\}$ of n d-dimensional boxes $I_i = ([\ell_{i,1}, r_{i,1}), [\ell_{i,2}, r_{i,2}), \ldots, [\ell_{i,d}, r_{i,d}))$ for $1 \leq i \leq n$.

For every point $x = (x_1, \ldots, x_d)$, let $S(x)$ be the set of boxes that include x, i. e., $S(x)$ contains all the elements I_i such that $\ell_{i,j} \leq x_j \leq r_{i,j}$ for all $1 \leq j \leq d$. For a coloring $\chi : \mathcal{I} \to K$, a color i, and a point x, let $c_i(x)$ be the number of boxes in $S(x)$ of color i. With the analog definition of imbalance $\text{imb}(\chi)$ and balance for d-dimensional boxes, the problem statement becomes:

MINIMUM IMBALANCE d-BOX k-COLORING

Instance: A set $\mathcal{I}$ of d-dimensional boxes.
Task: Find a k-coloring χ with minimal $\text{imb}(\chi)$.

First note that, unlike for the case $d = 1$, for $d \geq 2$ a balanced coloring may not exist: already for three rectangles, some instances (see Figure 4.3 for an

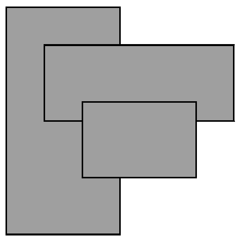


Figure 4.3: For $d = 2$ and three rectangles imbalance two is sometimes unavoidable.

example) require imbalance two. Hence, we also have a related decision problem:

BALANCED d-BOX k-COLORING

Instance: A set $\mathcal{I}$ of d-dimensional boxes.
Question: Is there a balanced k-coloring χ?

We show that for all $d \geq 2$ and $k \geq 2$, it is NP-complete to decide BALANCED d-BOX k-COLORING. This implies NP-hardness of MINIMUM IMBALANCE d-BOX k-COLORING.

Theorem 17. BALANCED d-BOX k-COLORING *is NP-complete for any* $d \geq 2$ *and any* $k \geq 2$.

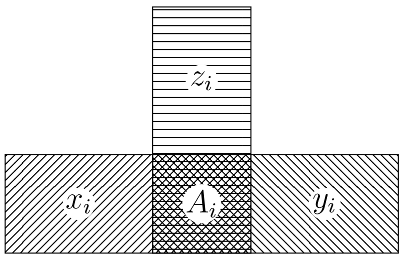
We will reduce from NOT-ALL-EQUAL 3SAT (NAE-3SAT) [48]. Note that the classic definition of NAE-3SAT [32] allows negated variables. However, this is not needed to make the problem NP-complete [48]. Thus, and for simplicity, in the sequel, we will assume that all variables occur only non-negatedly.

NOT-ALL-EQUAL 3SAT (NAE-3SAT)

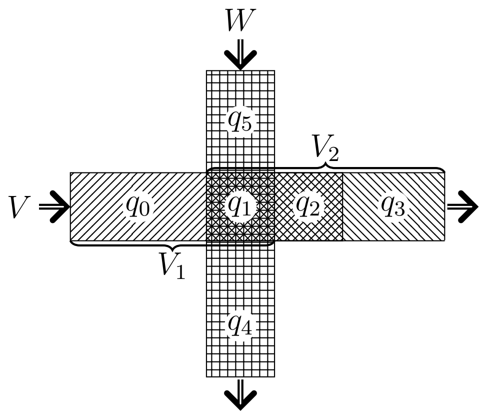
Instance: A Boolean formula with clauses $C_1, \ldots, C_m$, each having at most 3 variables.
Question: Is there a truth assignment such that in every clause, not all variables have the same value?

We first consider $k = 2$ and then generalize to arbitrary k. We present the gadgets of the reduction, then show how they are combined together, and conclude by proving correctness.

For each clause $C_i = (x_i, y_i, z_i)$, we construct a *clause gadget* comprised of three rectangles (see Figure 4.4(a)). Note that all three rectangles overlap in region A_i, and only there. Then we also construct a separate rectangle r_j for every



(a) Gadget for clause $C_i = (x_i, y_i, z_i)$



(b) Gadget for crossing chains V and W

Figure 4.4: The gadgets of the reduction.

variable. Finally, we connect each r_j to all rectangles that appear in a clause gadget, and correspond to the same variable as r_j. We do this by a chain with an odd number of rectangles. This ensures that in any balanced 2-coloring, r_j and the corresponding rectangle in the clause gadget have the same color. If two chains need to cross, we introduce a *crossing gadget* as seen in Figure 4.4(b). Three rectangles are relevant for the crossing of two chains V and W. The first is V_1 and contains areas q_0, q_1, and q_2, the second is V_2, containing q_1, q_2, and q_3. Both V_1 and V_2 belong to chain V. The last rectangle contains areas q_1, q_4 and q_5 and belongs to chain W. Note that the crossing does not induce any dependencies on the colorings between chains V and W. See Figure 4.5 for an example construction of an instance for BALANCED 2-BOX 2-COLORING.

Observe that the above construction only requires a number of rectangles polynomial in the size of the NAE-3SAT instance.

Lemma 26. BALANCED d-BOX 2-COLORING *is NP-complete for any* $d \geq 2$.

Proof. The problem is in NP, since feasibility of a color assignment can be verified in polynomial time. For NP-hardness, we show that a NAE-3SAT instance is satisfiable if and only if the answer to the corresponding BALANCED 2-BOX 2-COLORING instance is "yes". This also implies NP-completeness for every $d \geq 2$ by taking intervals of length 0 in higher dimensions.

($\Rightarrow$) Assume that there is a satisfying assignment of the NAE-3SAT instance. Then, color the rectangles r_j according to the truth values of their corresponding variables. This coloring can be easily extended to all the rectangles by alternatively coloring rectangles along a chain (and crossings) starting from each r_j and

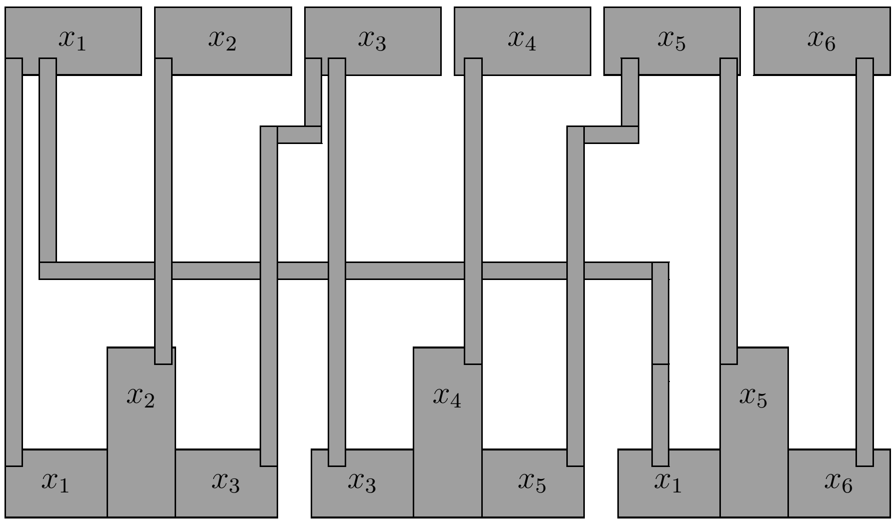


Figure 4.5: Example for NAE-3SAT instance (x_1, x_2, x_3), (x_3, x_4, x_5), (x_1, x_5, x_6).

ending at a clause gadget. It remains to show that $\mathrm{imb}(x) \leq 1$ holds for every point $x \in A_i$ for all $1 \leq i \leq m$. Consider A_i corresponding to clause $C_i = (x_i, y_i, z_i)$. The three rectangles that intersect at A_i have the colors corresponding to the truth values of their variables $x_i, y_i,$ and z_i in the solution of NAE-3SAT. Since the three variables do not have all the same truth value, the three rectangles cannot have all the same color, and $\mathrm{imb}(x) \leq 1$.

($\Leftarrow$) Assume that we have a balanced 2-coloring for the constructed BALANCED 2-BOX 2-COLORING instance. Consider only the clause gadgets. We have already observed that rectangles that correspond to the same variable and appear in clause gadgets must have the same color. We can assign the truth values of the variables according to the colors in the corresponding rectangles. Since in no A_i all three rectangles have the same color, in no C_i all three variables have the same truth value, yielding a feasible solution for NAE-3SAT. □

Proof of Theorem 17. First apply the construction for BALANCED 2-BOX 2-COLORING and call its rectangles *reduction rectangles*. Then add $k - 2$ additional rectangles that fully contain the construction and all intersect at least in one point outside the construction; these are called *cover rectangles*. By the latter property, cover rectangles must have distinct colors in any balanced coloring. Observe that each reduction rectangle contains some point that does not intersect with other reduction rectangles but only with all the cover rectangles. This implies that the reduction rectangles have available only the two colors not used by the cover rectangles. We conclude that the problem of k-coloring the constructed instance is equivalent to the problem of 2-coloring only the reduction rectangles. □

We consider a weighted version and show its NP-hardness by reduction from PARTITION. Furthermore, a variant with multiple intervals [35] is shown to be NP-hard by reduction from NAE-3SAT.

4.2.2 Multiple intervals

Another generalization suggested by Gyárfás and Lehel [35] is *multiple intervals*, where specific subsets of non-intersecting intervals must receive the same color. In our machine load balancing setting this can be seen as a job that is active only part of the time but still has to be assigned to one machine. This variant is also NP-complete. Given a NAE-3SAT instance where no negations are allowed, for every clause C_i construct three intervals corresponding to the three variables of the clause. All these three intervals have the same startpoints ℓ_i and endpoints r_i and $\ell_i > r_{i-1}$. Finally, for every variable, pack all the corresponding constructed intervals into a subset that enforces that they receive the same color. This is legal, since different intervals for the same variable are disjoint, and it can be easily seen that the multiple intervals instance has a balanced coloring if and only if the corresponding NAE-3SAT instance has a satisfying assignment.

Chapter 5

Discussion

In this thesis, we studied several scheduling problems with the objectives of minimizing energy consumption and keeping load balanced.

We first investigated the offline, classical deadline-based, energy conservation problem where a single variable-speed processor is equipped with a sleep state. In addition to settling the computational complexity of the optimization problem, and providing a lower bound for s_{crit}-schedules, we have developed algorithms achieving small approximation guarantees. All the algorithms use only one speed level, in addition to those of *YDS*. This is a positive feature because speed adjustments incur overhead in practice.

Although our approximation guarantees are small, the NP-hardness of the problem implies that an exact polynomial time algorithm for the problem is highly unlikely. The major open question remaining, is whether a polynomial time approximation scheme can be developed for speed scaling with sleep state.

Furthermore, the construction in the proof of our NP-hardness reduction implies that the problem is NP-hard even when the processor is only equipped with a sleep state along with two distinct and discrete speed levels. Studying the approximability of this problem is likely to provide useful insights for the more general problem. A PTAS for the problem where the processor has a constant number of speed levels along with the sleep state, would most likely lead to a PTAS for the general setting.

We then looked at the problem of speed scaling in multi-processor environments. We considered a classical scheduling problem where jobs have associated deadlines and assumed that job migration is allowed. For the offline problem, we have developed a combinatorial, polynomial time algorithm that efficiently computes optimal schedules. Furthermore we have extended the single processor online algorithms *Optimal Available* and *Average Rate*. Bansal et al. [19] gave an online algorithm for a single processor that, for large α, achieves a smaller competitiveness than *Optimal Available*. An open problem is if this strategy can also be extended to multi-processor systems.

Another working direction is to devise and analyze online algorithms for general convex power functions. Even for a single processor, no competitive strategy is known. Moreover, in the problem setting investigated in Chapter 2 the processor is equipped with an additional sleep state. It would be worthwhile to investigate combined speed scaling and power-down mechanisms in multi-processor environments. We denote that our proofs in the single-processor setting can probably not be extended, because not all properties of the critical speed translate to multiprocessor environments.

Finally, we focused on offline machine load balancing with identical machines, and the objective of minimizing the current load, where all jobs have unit weights. We reformulated the problem as a problem of coloring n intervals with k colors in a balanced way. We first have shown that a coloring with maximal difference at most one always exists, and developed a fast algorithm for finding such a coloring. Actually our result is more general: the polynomial time algorithm can be applied for k-coloring any hypergraph with the consecutive-ones property. This can be seen as a special case of k-coloring hypergraphs with a totally unimodular incidence matrix. An interesting extension would therefore be to study the problem on arbitrary totally unimodular incidence matrices. Furthermore, it might be worth trying to reduce the running time of our algorithm for k colors. A factor of k in the running time comes from the potentially large number of virtual intervals.

Another interesting open question, is how large the imbalance can become for d-dimensional boxes, and whether we can find polynomial-time approximations for it. We were not able to find an instance requiring an imbalance greater than 2 for the 2-dimensional case.

Bibliography

[1] Lesswatts.org: Race to idle. `http://www.lesswatts.org/projects/applications-power-management/race-to-idle.php`.

[2] Google details, and defends, its use of electricity. `http://www.nytimes.com/2011/09/09/technology/google-details-and-defends-its-use-of-electricity.html`.

[3] Udo Adamy and Thomas Erlebach. Online coloring of intervals with bandwidth. In *Proc. 1st International Workshop on Approximation and Online Algorithms (WAOA)*, volume 2909 of *LNCS*, pages 1–12. Springer, 2003.

[4] Geir Agnarsson and Magnús M. Halldórsson. Strong colorings of hypergraphs. In *Proc. 2nd International Workshop on Approximation and Online Algorithms (WAOA)*, volume 3351 of *LNCS*, pages 253–266. Springer, 2005.

[5] Susanne Albers. Energy-efficient algorithms. *Commun. ACM*, 53(5):86–96, 2010.

[6] Susanne Albers, Fabian Müller, and Swen Schmelzer. Speed scaling on parallel processors. In *Proc. 19th annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 289–298, 2007.

[7] Noga Alon. A simple algorithm for edge-coloring bipartite multigraphs. *Information Processing Letters*, 85(6):301–302, 2003.

[8] Matthew Andrews, Michel X. Goemans, and Lisa Zhang. Improved bounds for on-line load balancing. In *Proc. 2nd Annual International Conference on Computing and Combinatorics (COCOON)*, pages 1–10. Springer, 1996.

[9] Amitai Armon, Yossi Azar, and Leah Epstein. Temporary tasks assignment resolved. *Algorithmica*, 36(3):295–314, 2003.

[10] Giorgio Ausiello, Marco Protasi, Alberto Marchetti-Spaccamela, Giorgio Gambosi, Pierluigi Crescenzi, and Viggo Kann. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1999.

[11] Yossi Azar. On-line load balancing. In *Online Algorithms*, pages 178–195. Springer, 1996.

[12] Yossi Azar, Oded Regev, Jiri Sgall, and Gerhard J. Woeginger. Off-line temporary tasks assignment. *Theor. Comput. Sci.*, 287(2):419–428, 2002.

[13] Peter Bailis, Vijay Janapa Reddi, Sanjay Gandhi, David Brooks, and Margo Seltzer. Dimetrodon: processor-level preventive thermal management via idle cycle injection. In *Proc. 48th Design Automation Conference (DAC)*, pages 89–94. ACM, 2011.

[14] Nikhil Bansal. Constructive algorithms for discrepancy minimization. In *Proc. 51st Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 3–10, 2010.

[15] Nikhil Bansal, David P. Bunde, Ho-Leung Chan, and Kirk Pruhs. Average rate speed scaling. *Algorithmica*, 60:877–889, 2011.

[16] Nikhil Bansal, Ho-Leung Chan, Tak-Wah Lam, and Lap-Kei Lee. Scheduling for speed bounded processors. In *Proc. 35th International Colloquium on Automata, Languages and Programming (ICALP), Part I*, pages 409–420. Springer, 2008.

[17] Nikhil Bansal, Ho-Leung Chan, and Kirk Pruhs. Speed scaling with an arbitrary power function. In *Proc. 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 693–701. SIAM, 2009.

[18] Nikhil Bansal, Ho-Leung Chan, Kirk Pruhs, and Dmitriy Katz. Improved bounds for speed scaling in devices obeying the cube-root rule. In *Proc. 36th International Colloquium on Automata, Languages and Programming (ICALP), Part I*, pages 144–155. Springer, 2009.

[19] Nikhil Bansal, Tracy Kimbrel, and Kirk Pruhs. Speed scaling to manage energy and temperature. *Journal of the ACM*, 54:3:1–3:39, 2007.

[20] Philippe Baptiste. Scheduling unit tasks to minimize the number of idle periods: a polynomial time algorithm for offline dynamic power management. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 364–367. ACM, 2006.

[21] Philippe Baptiste, Marek Chrobak, and Christoph Dürr. Polynomial time algorithms for minimum energy scheduling. In *Proc. 15th Annual European Symposium on Algorithms (ESA)*, pages 136–150. Springer, 2007.

[22] J. Beck and T. Fiala. "Integer making" theorems. *Discrete Applied Mathematics*, 3(1):1–8, 1981.

[23] Therese C. Biedl, Eowyn Čenek, Timothy M. Chan, Erik D. Demaine, Martin L. Demaine, Rudolf Fleischer, and Ming-Wei Wang. Balanced k-colorings. *Discrete Mathematics*, 254(1–3):19–32, 2002.

[24] Brad D. Bingham and Mark R. Greenstreet. Energy optimal scheduling on multiprocessors with migration. In *Proc. 2008 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 153–161. IEEE Computer Society, 2008.

[25] Kellogg S. Booth and George S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computer and System Sciences*, 13(3):335–379, 1976.

[26] Ho-Leung Chan, Wun-Tat Chan, Tak-Wah Lam, Lap-Kei Lee, Kin-Sum Mak, and Prudence W. H. Wong. Energy efficient online deadline scheduling. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 795–804. SIAM, 2007.

[27] Richard Cole, Kirstin Ost, and Stefan Schirra. Edge-coloring bipartite multigraphs in $O(E \log D)$ time. *Combinatorica*, 21(1):5–12, 2001.

[28] Benjamin Doerr. *Integral Approximation*. Habilitationsschrift, Christian-Albrechts-Universität zu Kiel, 2005.

[29] Benjamin Doerr and Anand Srivastav. Multicolour discrepancies. *Combinatorics, Probability and Computing*, 12:365–399, 2003.

[30] Leah Epstein. Online interval coloring. In *Encyclopedia of Algorithms*, pages 594–598. Springer, 2008.

[31] Anshul Gandhi, Mor Harchol-Balter, Rajarshi Das, and Charles Lefurgy. Optimal power allocation in server farms. In *Proc. 11th International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 157–168. ACM, 2009.

[32] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[33] Matthew Garrett. Powering down. *Queue*, 5:16–21, November 2007.

[34] Gero Greiner, Tim Nonner, and Alexander Souza. The bell is ringing in speed-scaled multiprocessor scheduling. In *Proc. 21st Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 11–18. ACM, 2009.

[35] András Gyárfás and Jenő Lehel. Covering and coloring problems for relatives of intervals. *Discrete Mathematics*, 55(2):167–180, 1985.

[36] Xin Han, Tak-Wah Lam, Lap-Kei Lee, Isaac K. K. To, and Prudence W. H. Wong. Deadline scheduling and power management for speed bounded processors. *Theoretical Computer Science*, 411:3587–3600, 2010.

[37] Ian Holyer. The NP-completeness of edge-coloring. *SIAM Journal on Computing*, 10(4):718–720, 1981.

[38] Sandy Irani and Kirk R. Pruhs. Algorithmic problems in power management. *SIGACT News*, 36:63–76, 2005.

[39] Sandy Irani, Sandeep Shukla, and Rajesh Gupta. Algorithms for power savings. *ACM Transactions on Algorithms*, 3:1–23, 2007.

[40] Hal A. Kierstead and William T. Trotter. An extremal problem in recursive combinatorics. *Congressus Numerantium*, 33:143–153, 1981.

[41] Florian Kluge, Sascha Uhrig, Jörg Mische, Benjamin Satzger, and Theo Ungerer. Optimisation of energy consumption of soft real-time applications by workload prediction. In *Proc. 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*, pages 63–72. IEEE Computer Society, 2010.

[42] Dénes Kőnig. Gráfok és alkalmazásuk a determinánsok és a halmazok elméletére [in English: Graphs and their application to determinant theory and set theory]. *Matematikai és Természettudományi Értesítő*, 34:104–119, 1916.

[43] Tak-Wah Lam, Lap-Kei Lee, Isaac K. K. To, and Prudence W. H. Wong. Energy efficient deadline scheduling in two processor systems. In *Proc. 18th International Conference on Algorithms and Computation (ISAAC)*, pages 476–487. Springer, 2007.

[44] Minming Li, Becky Jie Liu, and Frances F. Yao. Min-energy voltage allocation for tree-structured tasks. *J. Comb. Optim.*, 11(3):305–319, 2006.

[45] Minming Li, Andrew C. Yao, and Frances F. Yao. Discrete and continuous min-energy schedules for variable voltage processors. In *Proceedings of the National Academy of Sciences USA*, volume 103, pages 3983–3987, 2006.

[46] Minming Li and Frances F. Yao. An efficient algorithm for computing optimal discrete voltage schedules. *SIAM Journal on Computing*, 35:658–671, 2005.

[47] Jiří Matoušek. *Geometric Discrepancy: An Illustrated Guide*, volume 18 of *Algorithms and Combinatorics*. Springer, 1999.

[48] Thomas J. Schaefer. The complexity of satisfiability problems. In *Proc. 10th Annual ACM Symposium on Theory of Computing (STOC)*, pages 216–226. ACM, 1978.

[49] Martin Skutella and José Verschae. A robust PTAS for machine covering and packing. In *Proc. 18th Annual European Symposium on Algorithms (ESA)*, pages 36–47, 2010.

[50] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28:202–208, 1985.

[51] Joel Spencer. Six standard deviations suffice. *Transactions of the American Mathematical Society*, 289(2):679–706, 1985.

[52] Aravind Srinivasan. Improving the discrepancy bound for sparse matrices: Better approximations for sparse lattice approximation problems. In *Proc. 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 692–701, 1997.

[53] Dominique de Werra. Equitable colorations of graphs. *Revue Française d'Informatique et de Recherche opérationnelle*, R-3:3–8, 1971.

[54] Jeffery Westbrook. Load balancing for response time. *J. Algorithms*, 35(1):1–16, 2000.

[55] Frances Yao, Alan Demers, and Scott Shenker. A scheduling model for reduced CPU energy. In *Proc. 36th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 374–382. IEEE Computer Society, 1995.

Erklärung

Hiermit erkläre ich,

- dass ich die vorliegende Dissertationsschrift „Scheduling Algorithms for Saving Energy and Balancing Load“ selbständig und ohne unerlaubte Hilfe angefertigt habe;
- ich mich nicht bereits anderwärts um einen Doktorgrad beworben habe oder einen solchen besitze, und
- mir die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt-Universität zu Berlin bekannt ist, gemäß amtliches Mitteilungsblatt Nr. 34/2006.

Berlin, den ....