Algorithms for Classical and Modern Scheduling Problems

Dissertation

zur Erlangung des Grades des
Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

vorgelegt von

Sebastian Ott

Saarbrücken
2016

Dekan der Naturwissenschaftlich-Technischen Fakultät I:
Prof. Dr. Frank-Olaf Schreyer

Prüfungsausschuss:
Prof. Dr. Raimund Seidel (Vorsitzender des Prüfungsausschusses)
Prof. Dr. Dr. h.c. mult. Kurt Mehlhorn
Prof. Dr. Susanne Albers
Priv.-Doz. Dr. Rob van Stee
Dr. Chien-Chung Huang
Dr. Thomas Kesselheim (Akademischer Beisitzer)

Tag des Kolloquiums:
4. Juli 2016

Abstract

Subject of this thesis is the design and the analysis of algorithms for scheduling problems. In the first part, we focus on energy-efficient scheduling, where one seeks to minimize the energy needed for processing certain jobs via dynamic adjustments of the processing speed (speed scaling). We consider variations and extensions of the standard model introduced by Yao, Demers, and Shenker in 1995 [79], including the addition of a sleep state, the avoidance of preemption, and variable speed limits.

In the second part, we look at classical makespan scheduling, where one aims to minimize the time in which certain jobs can be completed. We consider the restricted assignment model, where each job can only be processed by a specific subset of the given machines. For a special case of this problem, namely when heavy jobs can go to at most two machines, we present a combinatorial algorithm with approximation ratio strictly better than two.

Zusammenfassung

Inhalt dieser Arbeit ist der Entwurf und die Analyse von Algorithmen für Planungsprobleme. Der erste Teil konzentriert sich auf energieeffiziente Ablaufplanung, wobei bestimmte Aufträge mittels dynamischer Anpassungen der Arbeitsgeschwindigkeit (Speed Scaling) möglichst energiesparend abgearbeitet werden sollen. Es werden verschiedene Variationen und Erweiterungen des Standardmodells von Yao, Demers und Shenker aus dem Jahr 1995 [79] betrachtet, wie zum Beispiel die Hinzunahme eines Ruhezustands, die Vermeidung von Präemption, sowie variable Geschwindigkeitsbegrenzungen.

Im zweiten Teil geht es um ein klassisches Problem der Maschinenbelegungsplanung. Hier ist das Ziel bestimmte Aufträge mit minimalem Zeitaufwand abzuarbeiten. Betrachtet wird das Modell der eingeschränkten Zuordnung, bei dem jeder Auftrag nur auf einer Teilmenge der gegebenen Maschinen bearbeitet werden kann. Für einen Spezialfall dieses Problems, nämlich wenn große Aufträge auf höchstens zwei verschiedenen Maschinen bearbeitet werden können, wird ein kombinatorischer Algorithmus mit Approximationsgüte besser als zwei vorgestellt.

Acknowledgements

I am deeply grateful for the opportunity to do my PhD in the Algorithms and Complexity Department of the Max Planck Institute for Informatics, headed by Kurt Mehlhorn. A special thanks goes to my advisor Chien-Chung for bringing out the best in me. I greatly enjoyed our thrilling discussions and will never forget working with you.

During my PhD, I had the pleasure of working with many people: Fidaa Abed, Antonios Antoniadis, Chien-Chung Huang, Peter Kling, Sören Riechers, Rob van Stee, and Jose Verschae. I thank all of them.

Last but not least, I would like to express my deepest gratitude to my parents and my wife Natalie for their steady support.

"As a man who has devoted his whole life to the most clearheaded science, to the study of matter, I can tell you as a result of my research about the atoms this much: There is no matter as such! All matter originates and exists only by virtue of a force which brings the particles of an atom to vibration and holds this most minute solar system of the atom together. ... We must assume behind this force the existence of a conscious and intelligent Mind. This Mind is the matrix of all matter."

Max Planck[1]

[1] *Das Wesen der Materie (The Nature of Matter)*, Florence, Italy (1944). Archiv zur Geschichte der Max-Planck-Gesellschaft, Abt. Va, Rep. 11 Planck, Nr. 1797. Excerpt from Gregg Braden. *The Spontaneous Healing of Belief*, page 212. Hay House, 2008.

Contents

Introduction **1**

I. ENERGY-EFFICIENT SCHEDULING **5**

1. Speed Scaling with Sleep State **7**
1.1. Introduction . . . 7
1.1.1. Previous Work . . . 8
1.1.2. Our Contribution . . . 9
1.2. Preliminaries . . . 10
1.3. Discretizing the Problem . . . 14
1.3.1. Further Definitions and Notation . . . 15
1.3.2. Existence of a Near-Optimal Discretized Schedule . . . 17
1.4. The Dynamic Program . . . 23

2. Non-Preemptive Speed Scaling **25**
2.1. Introduction . . . 25
2.1.1. Our Results and Techniques . . . 25
2.1.2. Related Work . . . 27
2.1.3. Overview . . . 27
2.2. Preliminaries and Notations . . . 28
2.3. Laminar Instances . . . 31
2.4. Equal-Volume Jobs . . . 37
2.5. Purely-Laminar Instances . . . 38
2.6. Bounded Number of Time Windows . . . 40

3. Speed Scaling with Variable Electricity Rates and Speed Limits **43**
3.1. Introduction . . . 43
3.2. Model and Preliminaries . . . 44
3.3. Balance for Optimality . . . 45
3.3.1. Scheduling via Variational Calculus . . . 46
3.3.2. Characterizing Optimal Solutions . . . 48
3.3.3. Extracting Structural Properties . . . 51
3.4. Algorithm and Analysis . . . 56
3.4.1. Algorithm Description . . . 57
3.4.2. Correctness and Runtime . . . 58

II. MAKESPAN SCHEDULING 61

4. Graph Balancing with Light Hyper Edges 63

4.1. Introduction 63
4.2. Preliminaries 67
4.3. The 2-Valued Case 68
4.4. The General Case 75
4.4.1. Formal description of the algorithm 77
4.4.2. Proof of Lemma 4.4.4 81
4.4.3. Proof of Lemma 4.4.5 88

Bibliography 103

List of Figures

1.1. The time points defined for a particular zone 16

2.1. Time intervals of a laminar instance, arranged in a tree structure . . 31
2.2. Procedure for computing the DP-entries 35

3.1. Decreasing energy through moving workload 54
3.2. A work-balanced schedule 56

4.1. The procedure EXPLORE1 69
4.2. An example of a conflict set 75
4.3. A naive PUSH will oscillate the pebble 76
4.4. A fake orientation 76
4.5. The procedure FORCED ORIENTATIONS. 78
4.6. The procedure EXPLORE2. 79

Introduction

Scheduling decisions affect our lives in a significant way. Even an action as simple as buying a tube of toothpaste involves many small tasks that need to be scheduled: put on your shoes, get your keys and wallet, drive to a store, find toothpaste, queue up at the cashier, and so forth. A schedule is a precise plan how to carry out all these tasks, keeping in view the available resources. For instance, if you have no cash at home, you must fit in a stop at the bank before reaching the store.

In the most general sense, scheduling can be understood as the problem of assigning tasks to resources over a certain period of time. Today, the field of scheduling is a widely researched area with several hundred publications every year and early work dating back to at least the 1950s [68]. In one of the landmark papers of that time, Johnson [53] considers a production problem, where items must be processed through two different machines one after the other. Taking into account the setup- and work times for each item, Johnson designs a simple scheduling rule that minimizes the total elapsed time until all jobs are completed (the so-called *makespan*).

The problem considered by Johnson is a classical example of scheduling with a time-objective. For many years, *time* was the key benchmark to determine how efficient a solution is. In the second part of this thesis, we shall also consider a scheduling problem with makespan objective (Chapter 4) - a *classical* scheduling problem.

In recent years, however, the focus has begun to shift. No longer is faster always better. Instead, *energy*-efficiency has risen to be a major factor in the design and development of technical systems, ranging from data centers, over the embedded systems in our homes ("Internet of Things"), to the mobile devices everyone carries around. There are efforts on a multitude of levels to make such systems more energy-efficient. On an algorithmic level, these efforts focus largely on the *speed scaling* technique, which basically all modern microprocessors support in one way or the other. The general idea of this technique is to dynamically adjust the operating speed of the processor to the actual workload requirements. A higher speed implies a higher performance, but this performance comes at the cost of a higher power consumption. In practice, the power function $P(s) = s^3$ provides a good approximation on the true power consumption when the processor is run at speed s [23, 30]. The theoretical investigation of scheduling problems involving the speed scaling technique was initiated by Yao, Demers, and Shenker in their seminal paper of 1995 [79]. Their goal is to process a given set of jobs (each equipped with a release time, deadline, and processing volume) on a single speed-scalable processor, using as little energy as possible. For this problem they develop a polynomial-time algorithm, known as the YDS-algorithm. In the first part of this thesis, we de-

sign and analyze scheduling algorithms for variations and extensions of Yao et al.'s original model.

Basics on Approximation Algorithms

Most problems considered in this thesis belong to the class of NP-hard problems. It is widely believed that for this class of problems, optimal solutions cannot be computed efficiently (i.e. no algorithm with polynomial running time exists). The question whether this is true or not results in the famous *P versus NP* problem. We refer the interested reader to [42] for an extensive discussion on the theory of NP-completeness.

One possible way to deal with NP-hard problems is to develop *approximation algorithms*, that compute nearly optimal solutions in polynomial time. The quality of the returned solution is measured by the so-called *approximation ratio* α. We say that an algorithm is an α-approximation if, for a minimization problem, it returns a solution with objective function value at most α times the optimal. In the following, we will denote the optimal objective function value by OPT.

Sometimes it is possible to approximate a problem to any required degree, which leads us to the concept of *approximation schemes*. Consider an algorithm with two inputs: a problem instance and an error parameter $\epsilon > 0$. If the algorithm outputs a solution with objective function value at most $(1 + \epsilon) \cdot \mathrm{OPT}$ (for a minimization problem), it is called an approximation scheme. In a polynomial-time approximation scheme (PTAS), the algorithm's running time has to be polynomial in the size of the problem instance for each *fixed* $\epsilon > 0$. If the running time is also polynomial in $1/\epsilon$, it is called a fully polynomial-time approximation scheme (FPTAS).

A general introduction to the topic of approximation algorithms can be found in [75].

Outline of the Thesis

This thesis consists of two parts. In the first part we consider scheduling problems where minimizing energy consumption (or -cost) is the major goal. In the second part, we look at the classical makespan objective, i.e. we aim to minimize the time in which certain jobs can be completed. In the following, we give an overview of the different chapters. Note that each chapter is self-contained and can be read independently.

Chapter 1: In the first chapter, we study the scheduling of jobs in a computing environment not only equipped with speed scaling capabilities, but also with a so-called sleep state. In the sleep state, the processor consumes no energy, but a constant wake-up cost is required to transition back to the active state. In contrast to speed scaling alone, the addition of a sleep state makes it sometimes beneficial to accelerate the processing of jobs in order to transition the

processor to the sleep state for longer amounts of time and incur further energy savings. The goal is to output a feasible schedule that minimizes the energy consumption. Preemption of jobs is allowed. Since the introduction of the problem by Irani et al. [50], its exact computational complexity has been repeatedly posed as an open question (see e.g. [2, 27, 48]). Before our work, the best known upper and lower bounds were a 4/3-approximation algorithm and NP-hardness due to [2] and [2, 57], respectively.

We close the aforementioned gap between the upper and lower bound on the computational complexity of speed scaling with sleep state by presenting a fully polynomial-time approximation scheme for the problem.

Our result has been previously published in SODA '15 [10].

Chapter 2: In this chapter, we revisit the original speed scaling problem (without sleep state), as introduced by Yao, Demers, and Shenker in 1995 [79]. The goal is to process a given number of jobs (each with its own release time, deadline, and processing volume) on a single speed-scalable processor, using as little energy as possible. As opposed to the original setting, we consider the non-preemptive variant of the problem, where a job, once started, must be processed uninterruptedly until its completion. This version of the problem is significantly harder (it was shown to be strongly NP-hard by Antoniadis and Huang in 2012 [8]) and only a few (large) constant factor approximations exist [8, 16, 17, 34]. Up until now, the (general) complexity of this problem is unknown.

We study the special case when the jobs' time windows have a laminar structure. The problem remains strongly NP-hard in this case [8]. We present a quasipolynomial-time approximation scheme, thereby showing that (at least) this special case is not APX-hard, unless NP $\subseteq$ DTIME($2^{poly(\log n)}$).

Our second contribution is a polynomial-time exact algorithm for the case that all jobs have equal volume (even if the jobs' time windows do not have a laminar structure). In addition, we show that two other special cases of the problem allow fully polynomial-time approximation schemes.

The contents of this chapter have been previously published in MFCS '14 [46].

Chapter 3: In this chapter, we consider an extension of the (preemptive) speed scaling problem introduced by Yao et al. [79]: the maximum allowed speed of the processor is limited, and both this maximum speed and the energy costs may vary - even continuously - over time. The objective is to find a feasible schedule that minimizes the total energy costs.

We apply techniques from calculus of variations to derive optimality conditions that extend the well known KKT conditions to the case of continuous constraints. Using these optimality conditions, we design and analyze a natural and exact polynomial-time algorithm for this problem.

The contents of this chapter are joint work with Antonios Antoniadis (Max-Planck-Institut für Informatik, Saarbrücken, Germany), Peter Kling (Simon Fraser University, Burnaby, Canada), and Sören Riechers (University of Paderborn, Paderborn, Germany). Our result is currently under submission.

Chapter 4: In this chapter, we consider a classical machine scheduling problem, namely MAKESPAN MINIMIZATION IN RESTRICTED ASSIGNMENT. In a landmark paper in 1990 [58], Lenstra, Shmoys, and Tardos gave a 2-approximation algorithm and proved that the problem cannot be approximated within 1.5 unless P=NP. The upper and lower bounds of the problem have been essentially unimproved in the intervening 25 years, despite several remarkable successful attempts in some special cases of the problem [31, 37, 73] recently.

We consider a special case of this problem, called *graph-balancing with light hyper edges*, where heavy jobs can be assigned to at most two machines while light jobs can be assigned to any number of machines. For this case, we present algorithms with approximation ratios strictly better than 2.

Our algorithms are purely combinatorial, without the need of solving a linear program as required in most other known approaches.

The contents of this chapter are joint work with Chien-Chung Huang (Chalmers University, Gothenburg, Sweden) and are accepted for publication at ESA '16; a preprint is available on arXiv [47].

Part I.

ENERGY-EFFICIENT SCHEDULING

1. Speed Scaling with Sleep State

1.1. Introduction

As energy efficiency in computing environments becomes more and more important, chip manufacturers are increasingly incorporating energy-saving functionalities to their processors. One of the most common such functionalities is *dynamic speed scaling*, where the processor can adjust its speed dynamically over time. A lower speed results in a lower energy consumption, but at the cost of performance. However, even when the processor is idling, it consumes a non-negligible amount of energy just for the sake of "being active" (for example because of leakage current). Due to this fact, additional energy can be saved by incorporating a *sleep state* to the processor. A processor in a sleep state consumes zero (or negligible) energy; however, there is an extra energy cost when it is transitioned back to the active state.

In this chapter, we study the offline problem of minimizing energy consumption of a processor that is equipped with both speed scaling and sleep state capabilities. This problem is called *speed scaling with sleep state* and was first introduced by Irani et al. [50].

Let us state the problem more formally. The given processor has two states: the *active state*, during which it can execute jobs and consumes a certain amount of energy, and the *sleep state*, during which no jobs can be executed but also no energy is consumed. We assume that a *wake-up operation*, that is a transition from the sleep state to the active state, incurs a constant energy cost $C > 0$, whereas transitioning from the active state to the sleep state is free of charge. Further, as in [2, 50], the power required by the processor in the active state is an arbitrary convex and non-decreasing function P of its speed s. We assume that $P(0) > 0$, since (i) as already mentioned, real-world processors are known to have leakage current and (ii) otherwise the sleep state would be redundant. Further motivation for considering arbitrary convex power functions for speed scaling can be found, for example, in [24].

The input is a set $\mathcal{J}$ of n jobs. Each job j is associated with a release time r_j, a deadline d_j and a processing volume v_j. One can think of the processing volume as the number of CPU cycles that are required in order to completely process the job, so that if job j is processed at a speed of s, then v_j/s time-units are required to complete the job. We call the interval $[r_j, d_j)$ the *allowed interval* of job j, and say that job j is active at time point t if and only if $t \in [r_j, d_j)$[1]. Furthermore, we may assume without loss of generality that $\min_{j\in\mathcal{J}} r_j = 0$, and that $v_{min} := \min_{j\in\mathcal{J}} v_j$ is

[1] Unless stated differently, throughout this chapter an interval will always have the form $[\cdot,\cdot)$.

normalized to 1 (if $v^* \neq 1$ is the real minimum volume, we can scale the instance by dividing the r_j's, d_j's, and v_j's by v^*, and using the power function $P(s) \cdot v^*$ along with the original wake-up cost C). Further, let $d_{max} := \max_{j \in \mathcal{J}} d_j$ be the latest deadline of any job.

A *schedule* is defined as a mapping of every time point t to the state of the processor, its speed, and the job being processed at t (or *null* if there is no job running at t). Note that the processing speed is zero whenever the processor sleeps, and that a job can only be processed when the speed is strictly positive. A schedule is called *feasible* when the whole processing volume of every job j is completely processed in j's allowed interval $[r_j, d_j)$. Preemption of jobs is allowed.

The energy consumption incurred by a schedule $\mathcal{S}$ while the processor is in the active state, is its power integrated over time, i.e. $\int P(s(t))dt$, where $s(t)$ is the processing speed at time t, and the integral is taken over all time points in $[0, d_{max})$ during which the processor is active under $\mathcal{S}$. Assume that $\mathcal{S}$ performs k transitions from the sleep state to the active state. (We will assume that initially, prior to the first release time, as well as finally, after the last deadline, the processor is in the active state. However, our results can be easily adapted for the setting where the processor is initially and/or eventually in the sleep state). Then the total energy consumption of $\mathcal{S}$ is $E(\mathcal{S}) := \int P(s(t))dt + kC$, where again the integral is taken over all time points at which $\mathcal{S}$ keeps the processor in the active state. We are seeking a feasible schedule that minimizes the total energy consumption.

Observe that, by Jensen's inequality, and by the convexity of the power function, it is never beneficial to process a job with a varying speed. Irani et al. [50] observed the existence of a *critical speed* s_{crit}, which is the most efficient speed for processing jobs. This critical speed is the smallest speed that minimizes the function $P(s)/s$. Note that, by the convexity of $P(s)$, the only case where the critical speed s_{crit} is not well defined, is when $P(s)/s$ is always decreasing. However, this would render the setting unrealistic, and furthermore make the algorithmic problem trivial, since it would be optimal to process every job at an infinite speed. We may therefore assume that this case does not occur. Further, it can be shown (see [50]) that for any $s \geq s_{crit}$, the function $P(s)/s$ is non-decreasing.

1.1.1. Previous Work

The theoretical model for dynamic speed scaling was introduced in a seminal paper by Yao, Demers and Shenker [79]. They developed a polynomial time algorithm called *YDS*, that outputs a minimum-energy schedule for this setting. Irani, Shukla and Gupta [50] initiated the algorithmic study of speed scaling combined with a sleep state. Such a setting motivates the so-called *race-to-idle* technique: one saves energy by accelerating some jobs in order to transition the processor to the sleep state for longer periods of time (see [13, 41, 43, 69] and references therein for more information regarding the race-to-idle technique). Irani et al. developed a 2-approximation algorithm for speed scaling with sleep state, but the computational complexity of the problem has remained open. The first step towards attacking this open problem

was made by Baptiste [26], who gave a polynomial time algorithm for the special case where the processor must execute all jobs at a fixed speed, and all jobs are of unit size. Baptiste's algorithm is based on a clever dynamic programming formulation of the scheduling problem, and was later extended to (i) arbitrarily-sized jobs in [27], and (ii) a multiprocessor setting in [35].

More recently, Albers and Antoniadis [2] improved the upper bound on the approximation ratio of the general problem, by developing a 4/3-approximation algorithm. For the special case of agreeable deadlines and a power function of the form $P(s) = s^\alpha + \beta$ (with constant $\alpha > 1$ and $\beta > 0$), Bampis et al. [14] provided an exact polynomial time algorithm. With respect to the lower bound, [2] gave an NP-hardness reduction from the *partition* problem. The reduction uses a particular power function that is based on the partition instance, i.e., it is considered that the power function is part of the input. The reduction of [2] was later refined by Kumar and Shannigrahi [57], to show that the problem is NP-hard for any fixed, non-decreasing and strictly convex power function.

The online setting of the problem has also been studied. Irani et al. [50] gave a $(2^{2\alpha-2}\alpha^\alpha + 2^{\alpha-1} + 2)$-competitive online algorithm. Han et al. [45] improved upon this result by developing an $(\alpha^\alpha + 2)$-competitive algorithm for the problem. Both of the above results assume a power function of the form $P(s) = s^\alpha + \beta$, where $\alpha > 1$ and $\beta > 0$ are constants.

A more thorough discussion on the above scheduling problems can be found in the surveys [1, 48].

1.1.2. Our Contribution

We study the offline setting of speed scaling with sleep state. Since the introduction of the problem by Irani et al. [50], its exact computational complexity has been repeatedly posed as an open question (see e.g. [2, 27, 48]). Before our work, the best known upper and lower bounds were a 4/3-approximation algorithm and NP-hardness due to [2] and [2, 57], respectively. We settle the open question regarding the computational complexity of the problem by presenting a fully polynomial-time approximation scheme.

At the core of our approach is a transformation of the original preemptive problem into a non-preemptive scheduling problem of the same type. At first sight, this may seem counterintuitive, especially as Bampis et al. [15] showed that (for the problem of speed scaling alone) the ratio of an optimal preemptive solution against an optimal non-preemptive solution can be very high. However, this does not apply in our case, as we consider the non-preemptive problem on a modified instance, where each job is replaced by a polynomial number of *pieces*. Furthermore, in our analysis, we make use of a particular lexicographic ordering that does exploit the advantages of preemption.

In order to compute an optimal schedule for the modified instance via dynamic programming, we require a number of properties that pieces must satisfy in a valid schedule. The definition of these properties is based on a discretization of the time

horizon by a polynomial number of time points. Roughly speaking, we focus on those schedules that start and end the processing of each piece at such time points, and satisfy a certain constraint on the processing order of the pieces. Proving that a near-optimal schedule in this class of schedules exists is the most subtle part of our approach.

On the one hand, the processing order constraint can be exploited by the DP; on the other hand, such a constraint is difficult to establish in an optimal schedule with the introduced indivisible volumes (since pieces of different jobs might have different volumes and cannot easily be interchanged). To get around this, we first ensure the right ordering in an optimal schedule for the preemptive setting, and then perform a series of transformations to a non-preemptive schedule with the above properties. Each of these transformations increases the energy consumption only by a small factor, and maintains the correct ordering among the pieces.

We remark that Baptiste [26] used a dynamic program of similar structure for the special case of unit-sized jobs and a fixed-speed processor equipped with a sleep state. His dynamic program is also based on a particular ordering of jobs, which, however, is not sufficient for our setting. Since we have pieces of different sizes, the swapping argument used in [26] fails.

In Section 1.2, we describe the YDS algorithm of [79] for the problem of speed scaling *without* a sleep state, and then show several properties that a schedule produced by YDS has for our problem of speed scaling with sleep state. We then, in Section 1.3, define a particular class of schedules that have a set of desirable properties, and show that there exists a schedule in this class, whose energy consumption is within a $(1+\epsilon)$-factor from optimal. Finally, in Section 1.4, we develop an algorithm based on a dynamic program, that outputs, in polynomial time, a schedule of minimal energy consumption among all the schedules of the aforementioned class.

1.2. Preliminaries

We start by giving a short description of the YDS algorithm presented in [79]. For any interval I, let $B(I)$ be the set of jobs whose allowed intervals are contained in I. We define the *density* of I as

$$dens(I) = \frac{\sum_{j \in B(I)} v_j}{|I|}.$$

Note that the average speed that any feasible schedule uses during interval I is no less than $dens(I)$. YDS works in rounds. In the first round, the interval I_1 of maximal density is identified, and all jobs in $B(I_1)$ are scheduled during I_1 at a speed of $dens(I_1)$, according to the earliest deadline first policy. Then the jobs in $B(I_1)$ are removed from the instance and the time interval I_1 is "blacked out". In general, during round i, YDS identifies the interval I_i of maximal density (while disregarding the blacked out time intervals, and the already scheduled jobs), and then processes all jobs in $B(I_i)$ at a uniform speed of $dens(I_i)$. YDS terminates when all jobs are scheduled, and its running time is polynomial in the input size.

We remark that the speed used for the processing of jobs can never increase between two consecutive rounds, i.e., YDS schedules the jobs by order of non-increasing speeds. Furthermore, all the jobs scheduled in each round i have their allowed intervals within I_i.

Given any job instance $\mathcal{J}$, let $FAST(\mathcal{J})$ be the subset of $\mathcal{J}$ that YDS processes at a speed greater than or equal to s_{crit}, and let $SLOW(\mathcal{J}) := \mathcal{J} \setminus FAST(\mathcal{J})$. The following lemma is an extension of a fact proven by Irani et al. [50].

Lemma 1.2.1. *For any job instance $\mathcal{J}$, there exists an optimal schedule (w.r.t. speed scaling with sleep state) in which*

1. *Every job in $FAST(\mathcal{J})$ is processed according to YDS.*
2. *Every job $k \in SLOW(\mathcal{J})$ is run at a uniform speed $s_k \leq s_{crit}$, and the processor never (actively) runs at a speed less than s_k during $[r_k, d_k)$.*

We call an optimal schedule with these properties a YDS-extension *for $\mathcal{J}$.*

Proof. To break ties among schedules with equal energy consumption, we introduce the pseudo cost function $\int s(t)^2 dt$ (this idea was first used in [50]). Consider a minimal pseudo cost schedule Y, so that Y satisfies property 1 of the lemma, and minimizes the energy consumption among all schedules satisfying this property. It was shown in [50] that Y is optimal for instance $\mathcal{J}$, and that under Y

> Every job $k \in SLOW(\mathcal{J})$ is run at a uniform speed s_k, and the processor never (actively) runs at a speed less than s_k during those portions of $[r_k, d_k)$ where no job from FAST($\mathcal{J}$) is processed. $(*)$

It therefore remains to prove that the speeds s_k are no higher than s_{crit}. For the sake of contradiction, assume that there exists a job $j \in SLOW(\mathcal{J})$ which is processed at speed higher than s_{crit}. Let $\mathcal{I}$ be a maximal time interval, so that (i) $\mathcal{I}$ includes at least part of the execution of j, and (ii) at any time point $t \in \mathcal{I}$ the processor either runs strictly faster than s_{crit}, or executes a job from $FAST(\mathcal{J})$. Then there must exist a job $k \in SLOW(\mathcal{J})$ (possibly $k = j$) which is executed to some extent during $\mathcal{I}$, and whose allowed interval is not contained in $\mathcal{I}$ (otherwise, when running YDS, the density of $\mathcal{I}$ after the jobs in $FAST(\mathcal{J})$ have been scheduled is larger than s_{crit}, contradicting the fact that YDS processes all remaining jobs slower than s_{crit}). By the maximality of $\mathcal{I}$, there exists some interval $\mathcal{I}' \subseteq [r_k, d_k)$ right before $\mathcal{I}$ or right after $\mathcal{I}$, during which no job from $FAST(\mathcal{J})$ is executed, and the processor either runs with speed at most s_{crit} or resides in the sleep state. The first case contradicts property $(*)$, as k is processed during $\mathcal{I}$ and thus at speed $s_k > s_{crit}$. In the second case, we can use a portion of $\mathcal{I}'$ to slightly slow down k to a new speed s', such that $s_{crit} < s' < s_k$. The resulting schedule Y' has energy consumption no higher than Y, as $P(s)/s$ is non-decreasing for $s \geq s_{crit}$. Furthermore, if $\mathcal{C}_p$ is the pseudo cost of Y, then Y' has pseudo cost $\mathcal{C}_p - v_k s_k + v_k s' < \mathcal{C}_p$. This contradicts our assumptions on Y. □

By the preceding lemma, we may use YDS to schedule the jobs in $FAST(\mathcal{J})$, and need to find a good schedule only for the remaining jobs (which are exactly $SLOW(\mathcal{J})$). To this end, we transform the input instance $\mathcal{J}$ to an instance $\mathcal{J}'$, in which the jobs $FAST(\mathcal{J})$ are replaced by dummy jobs. This introduction of dummy jobs bears resemblance to the approach of [2]. We then show in Lemma 1.2.3, that any schedule for $\mathcal{J}'$ with a certain property, can be transformed to a schedule for $\mathcal{J}$ without any degradation in the approximation factor.

Consider the schedule S_{YDS} that algorithm YDS produces on $\mathcal{J}$. Let $I_i = [y_i, z_i)$, $i = 1, \dots, \ell$ be the i-th maximal interval in which S_{YDS} continuously runs at a speed greater than or equal to s_{crit}, and let $T_1, \dots, T_m$ be the remaining maximal intervals in $[0, d_{max})$ not covered by intervals $I_1, I_2, \dots, I_\ell$. Furthermore, let $\mathcal{T} := \cup_{1 \le k \le m} T_k$. Note that the intervals I_i and T_i partition the time horizon $[0, d_{max})$, and furthermore, by the way YDS is defined, every job $j \in FAST(\mathcal{J})$ is active in exactly one interval I_i, and is not active in any interval T_i. On the other hand, a job $j \in SLOW(\mathcal{J})$ may be active in several (consecutive) intervals I_i and $T_{i'}$. We transform $\mathcal{J}$ to a job instance $\mathcal{J}'$ as follows:

- For every job $j \in SLOW(\mathcal{J})$, if there exists an i such that $r_j \in I_i$ (resp. $d_j \in I_i$), then we set $r_j := z_i$ (resp. $d_j := y_i$), else we keep the job as it is.

- For each I_i, we replace all jobs $j \in FAST(\mathcal{J})$ that are active in I_i by a single job j_i^d with release time at y_i, deadline at z_i, and processing volume v_i^d equal to the total volume that S_{YDS} schedules in I_i, i.e. $v_i^d = \sum_{j \in B(I_i)} v_j$.

Clearly, the above transformation can be done in polynomial time. Note that after the transformation, there is no release time or deadline in the interior of any interval I_i. Furthermore, we have the following proposition:

Proposition 1.2.2. *$FAST(\mathcal{J}') = \{j_i^d : 1 \le i \le \ell\}$ and $SLOW(\mathcal{J}') = SLOW(\mathcal{J})$.*

Proof. Since $\mathcal{J}' = \{j_i^d : 1 \le i \le \ell\} \cup SLOW(\mathcal{J})$, and furthermore $SLOW(\mathcal{J}')$ and $FAST(\mathcal{J}')$ are disjoint sets, it suffices to show that (i) $FAST(\mathcal{J}') \supseteq \{j_i^d : 1 \le i \le \ell\}$ and that (ii) $SLOW(\mathcal{J}') \supseteq SLOW(\mathcal{J})$.

For (i), we observe that no job j_i^d can be feasibly scheduled at a uniform speed less than s_{crit}. As YDS uses a uniform speed for each job, these jobs must belong to $FAST(\mathcal{J}')$.

For (ii), consider the execution of YDS on $\mathcal{J}'$. More specifically, consider the first round when a job from $SLOW(\mathcal{J})$ is scheduled. Let $\mathcal{I}$ be the maximal density interval of this round, and let $\mathcal{J}_S$ and $\mathcal{J}_d$ be the sets of jobs from $SLOW(\mathcal{J})$ and $\{j_i^d : 1 \le i \le \ell\}$, respectively, that are scheduled in this round (note that $\mathcal{I}$ contains the allowed intervals of these jobs). As the speed used by YDS is non-increasing from round to round, it suffices to show that $dens(\mathcal{I}) < s_{crit}$.

Consider a partition of $\mathcal{I}$ into maximal intervals $\Lambda_1, \dots, \Lambda_a$, s.t. each Λ_k is con-

tained in some interval I_i or T_i. Then

$$\begin{aligned} dens(\mathcal{I}) &= \frac{\sum_{j\in\mathcal{J}_d} v_j}{|\mathcal{I}|} + \frac{\sum_{j\in\mathcal{J}_S} v_j}{|\mathcal{I}|} \\ &= \sum_{\Lambda_k \not\subseteq \mathcal{T}} \left(\frac{|\Lambda_k|}{|\mathcal{I}|} dens(\Lambda_k) \right) + \frac{\sum_{\Lambda_k\subseteq\mathcal{T}} |\Lambda_k|}{|\mathcal{I}|} \cdot \frac{\sum_{j\in\mathcal{J}_S} v_j}{\sum_{\Lambda_k\subseteq\mathcal{T}} |\Lambda_k|} \\ &\qquad\qquad \le \Big(\sum_{\Lambda_k \not\subseteq \mathcal{T}} \frac{|\Lambda_k|}{|\mathcal{I}|} \Big) dens(\mathcal{I}) + \Big(1 - \sum_{\Lambda_k \not\subseteq \mathcal{T}} \frac{|\Lambda_k|}{|\mathcal{I}|} \Big) \cdot \frac{\sum_{j\in\mathcal{J}_S} v_j}{\sum_{\Lambda_k\subseteq\mathcal{T}} |\Lambda_k|}, \end{aligned}$$

since no Λ_k can have a density larger than $dens(\mathcal{I})$ (because $\mathcal{I}$ is the interval of maximal density). It follows that

$$dens(\mathcal{I}) \le \frac{\sum_{j\in\mathcal{J}_S} v_j}{\sum_{\Lambda_k\subseteq\mathcal{T}} |\Lambda_k|} \ .$$

Furthermore, by the definition of $SLOW(\mathcal{J})$, it is possible to schedule all jobs in $\mathcal{J}_S$ during $\mathcal{I} \cap \mathcal{T}$, at a speed slower than s_{crit} (since none of the steps in the transformation from $\mathcal{J}$ to $\mathcal{J}'$ reduces the time any job is active during $\mathcal{T}$). Together with the previous inequality, this implies $dens(\mathcal{I}) < s_{crit}$. □

The following lemma suggests that for obtaining an FPTAS for instance $\mathcal{J}$, it suffices to give an FPTAS for instance $\mathcal{J}'$, as long as we schedule the jobs j_i^d exactly in their allowed intervals I_i.

Lemma 1.2.3. *Let S' be a schedule for input instance $\mathcal{J}'$, that (i) processes each job j_i^d exactly in its allowed interval I_i (i.e. from y_i to z_i), and (ii) is a c-approximation for $\mathcal{J}'$. Then S' can be transformed in polynomial time into a schedule S that is a c-approximation for input instance $\mathcal{J}$.*

Proof. Given such a schedule S', we leave the processing in the intervals $T_1, \dots, T_m$ unchanged, and replace for each interval I_i the processing of job j_i^d by the original YDS-schedule S_{YDS} during I_i. It is easy to see that the resulting schedule S is a feasible schedule for $\mathcal{J}$. We now argue about the approximation factor.

Let OPT be a YDS-extension for $\mathcal{J}$, and let OPT′ be a YDS-extension for $\mathcal{J}'$. Recall that $E(\cdot)$ denotes the energy consumption of a schedule (including wake-up costs). Additionally, let $E^I(S)$ denote the total energy consumption of S in all intervals $I_1, \dots, I_\ell$ without wake-up costs (i.e. the energy consumption for processing or being active but idle during those intervals), and define similarly $E^I(S')$, $E^I(OPT)$, and $E^I(OPT')$ for the schedules S', OPT, and OPT′, respectively. Since S' is a c-approximation for $\mathcal{J}'$, we have

$$E(S') \le cE(OPT').$$

Note that OPT′ schedules exactly the job j_i^d in each I_i (using the entire interval for it) by Proposition 1.2.2, and thus each of the schedules S, S', OPT, and OPT′ keeps

the processor active during every entire interval I_i. Therefore

$$E(S) - E(S') = E^I(S) - E^I(S'),$$

since S and S' have the same wake-up costs and do not differ in the intervals $T_1, \ldots, T_m$. Moreover,

$$E(OPT) - E(OPT') = E^I(OPT) - E^I(OPT'),$$

as $E(OPT) - E^I(OPT)$ and $E(OPT') - E^I(OPT')$ are both equal to the optimal energy consumption during $\mathcal{T}$ of any schedule that processes the jobs $SLOW(\mathcal{J})$ in $\mathcal{T}$ and resides in the active state during each interval I_i (including all wake-up costs of the schedule). Clearly, $E^I(S) = E^I(OPT)$, and since both S' and OPT$'$ schedule exactly the job j_i^d in each I_i (using the entire interval for it), we have that $E^I(S') \geq E^I(OPT')$. Therefore

$$E(S) - E(S') \leq E(OPT) - E(OPT').$$

We next show that $0 \leq E^I(OPT) - E^I(OPT') = E(OPT) - E(OPT')$, which implies

$$\begin{aligned} E(S) &\leq E(OPT) - E(OPT') + E(S') \\ &\leq c(E(OPT) - E(OPT')) + E(S') \\ &\leq c(E(OPT) - E(OPT')) + cE(OPT') \\ &\leq cE(OPT). \end{aligned}$$

Since YDS (when applied to $\mathcal{J}$) processes a volume of exactly v_i^d in each interval I_i, the average speed of OPT in I_i is $v_i^d/|I_i|$. On the other hand, OPT$'$ runs with a speed of exactly $v_i^d/|I_i|$ during I_i, and therefore $E^I(OPT) \geq E^I(OPT')$. □

1.3. Discretizing the Problem

After the transformation in the previous section, we have an instance $\mathcal{J}'$. In this section, we show that there exists a “discretized” schedule for $\mathcal{J}'$, whose energy consumption is at most $1 + \epsilon$ times that of an optimal schedule for $\mathcal{J}'$. In the next section, we will show how such a discretized schedule can be found by dynamic programming.

Before presenting formal definitions and technical details, we here first sketch the ideas behind our approach.

A major challenge of the original problem is that we need to deal with an infinite number of possible schedules. We overcome this intractability by “discretizing” the problem as follows: (1) we break each job in $SLOW(\mathcal{J}')$ into smaller pieces, and (2) we create a set of time points and introduce the additional constraint that each piece of a job has to start and end at these time points. The number of the introduced time points and job pieces are both polynomial in the input size and $1/\epsilon$, which greatly limits the amount of guesswork we have to do in the dynamic program. The challenge is how to find such a discretization and argue that it does not increase the optimal energy consumption by too much.

1.3.1. Further Definitions and Notation

We first define the set W of time points. Given an error parameter $\epsilon > 0$, let $\delta := \min\{\frac{1}{4}, \frac{\epsilon}{4}\frac{P(s_{crit})}{P(2s_{crit})-P(s_{crit})}\}$. Intuitively, δ is defined in such a way that speeding up the processor by a factor $(1+\delta)^3$ does not increase the power consumption by more than a factor $1+\epsilon$ (see Lemma 1.3.7).

Let $W' := \bigcup_{j\in\mathcal{J}'}\{r_j, d_j\}$, and consider the elements of W' in sorted order. Let $t_i, 1 \le i \le |W'|$ be the i-th element of W' in this order. We call an interval $[t_i, t_{i+1})$ for $1 \le i \le |W'|-1$ a *zone*, and observe that every zone is either equal to some interval I_i or contained in some interval T_i.

For each i in $1,\ldots,|W'|-1$, let $x(i)$ be the largest integer j so that

$$(1+\delta)^j \frac{1}{4n^2 s_{crit}(1+\delta)\lceil 1/\delta\rceil} \le t_{i+1} - t_i.$$

We are now ready to define the set W of time points as follows:

$$W := W' \bigcup_{\substack{i \text{ s.t. } [t_i,t_{i+1})\subseteq\mathcal{T}\\ 0\le j\le x(i)\\ 1\le r\le 16n^6\lceil 1/\delta\rceil^2(1+\lceil 1/\delta\rceil)}} \left\{ t_i + r\cdot\frac{(1+\delta)^j\frac{1}{4n^2 s_{crit}(1+\delta)\lceil 1/\delta\rceil}}{16n^6\lceil 1/\delta\rceil^2(1+\lceil 1/\delta\rceil)},\; t_{i+1} - r\cdot\frac{(1+\delta)^j\frac{1}{4n^2 s_{crit}(1+\delta)\lceil 1/\delta\rceil}}{16n^6\lceil 1/\delta\rceil^2(1+\lceil 1/\delta\rceil)} \right\}.$$

Let us explain how these time points in W come about. As we will show later (Lemma 1.3.5(2)), there exists a certain optimal schedule for $\mathcal{J}'$ in which each zone $[t_i, t_{i+1}) \subseteq \mathcal{T}$ contains at most one contiguous maximal processing interval, and this interval "touches" either t_i or t_{i+1} (or both). The geometric series $(1+\delta)^j\frac{1}{4n^2 s_{crit}(1+\delta)\lceil 1/\delta\rceil}$ of time points is used to approximate the ending/starting time of this maximal processing interval. For each guess of the ending/starting time, we split the guessed interval, during which the job pieces (to be defined formally immediately) are to be processed, into $16n^6\lceil 1/\delta\rceil^2(1+\lceil 1/\delta\rceil)$ many intervals of equal length. An example of the set W for a given zone can be seen in Figure 1.1.

Note that $|W|$ is polynomial in the input size and $1/\epsilon$.

Definition 1.3.1. *We split each job $j \in SLOW(\mathcal{J}')$ into $4n^2\lceil 1/\delta\rceil$ equal sized* pieces, *and also consider each job $j_i^d \in FAST(\mathcal{J}')$ as a single piece on its own. For every piece u of some job j, let $job(u) := j$, $r_u := r_j$, $d_u := d_j$, and $v_u := v_j/(4n^2\lceil 1/\delta\rceil)$ if $j \in SLOW(\mathcal{J}')$, and $v_u := v_j$ otherwise. Furthermore, let D denote the set of all pieces derived from all jobs in $\mathcal{J}'$.*

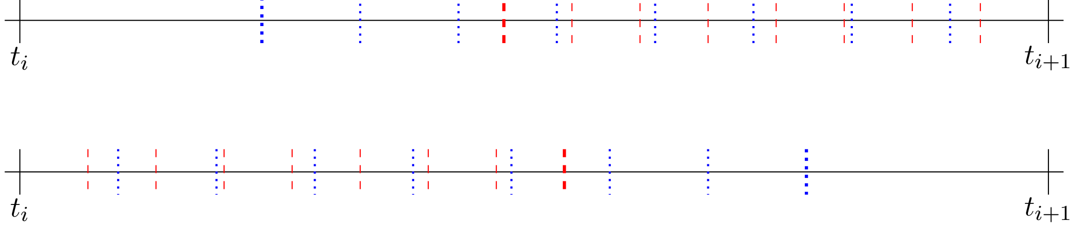


Figure 1.1.: We assume that $r = 1 \dots 8$ and that $x(i) = 2$. The red dashed points correspond to $j = 1$ and the blue dotted points to $j = 2$. For clarity, we drew the points defined from t_i and from t_{i+1} in two separate pictures. Note that for each j the number of points is the same and the points of the same color are at equal distance from each other.

Note that $|D| = \ell + |SLOW(\mathcal{J}')| \cdot 4n^2\lceil 1/\delta \rceil$ is polynomial in the input size and $1/\epsilon$. We now define an ordering of the pieces in D.

Definition 1.3.2. *Fix an arbitrary ordering of the jobs in $\mathcal{J}'$, s.t. for any two different jobs j and j', $j \prec j'$ implies $r_j \leq r_{j'}$. Now extend this ordering to the set of pieces, s.t. for any two pieces u and u', there holds*

$$u \prec u' \Rightarrow job(u) \preceq job(u').$$

We point out that any schedule for $\mathcal{J}'$ can also be seen as a schedule for D, by implicitly assuming that the pieces of any fixed job are processed in the above order.

We are now ready to define the class of discretized schedules.

Definition 1.3.3. *A* discretized schedule *is a schedule for $\mathcal{J}'$ that satisfies the following two properties:*

(i) Every piece is completely processed in a single zone, and without preemption.

(ii) The execution of every piece starts and ends at a time point from the set W.

A discretized schedule S is called well-ordered *if and only if*

(iii) For any time point t, such that in S a piece u ends at t, S schedules all pieces $u' \succ u$ with $d_{u'} \geq t$ after t.

Finally, we define a particular ordering over possible schedules, which will be useful in our analysis.

Definition 1.3.4. *Consider a given schedule. For every job $j \in \mathcal{J}'$, and every $x \leq v_j$, let $c_j(x)$ denote the earliest time point at which volume x of job j has been finished under this schedule. Furthermore, for any $j \in \mathcal{J}'$, we define*

$$q_j := \int_0^{v_j} c_j(x)dx.$$

Let $j_1 \prec j_2 \prec \cdots \prec j_{|\mathcal{J}'|}$ be the jobs in $\mathcal{J}'$. A schedule S is lexicographically smaller *than a schedule S' if and only if it is lexicographically smaller with respect to the vector $(q_{j_1}, q_{j_2}, \dots, q_{j_{|\mathcal{J}'|}})$.*

Observe that shifting the processing interval of any fraction of some job j to an earlier time point (without affecting the other processing times of j) decreases the value of q_j.

1.3.2. Existence of a Near-Optimal Discretized Schedule

In this section, we first show that there exists a YDS-extension for $\mathcal{J}'$ with certain nice properties (recall that a YDS-extension is an optimal schedule satisfying the properties of Lemma 1.2.1). We then explain how such a YDS-extension can be transformed into a well-ordered discretized schedule, and prove that the speed of the latter, at all times, is at most $(1+\delta)^3$ times that of the former. This fact essentially guarantees the existence of a well-ordered discretized schedule with energy consumption at most $1+\epsilon$ that of an optimal schedule for $\mathcal{J}'$.

Lemma 1.3.5. *Let OPT be a lexicographically minimal YDS-extension for $\mathcal{J}'$. Then the following holds:*

1. *Every job j_i^d is scheduled exactly in its allowed interval I_i.*
2. *Every zone $[t_i, t_{i+1}) \subseteq \mathcal{T}$ has the following two properties:*
   - *(a) There is at most one contiguous maximal processing interval within $[t_i, t_{i+1})$, and this interval either starts at t_i and/or ends at t_{i+1}. We call this interval the* block *of zone $[t_i, t_{i+1})$.*
   - *(b) OPT uses a uniform speed of at most s_{crit} during this block.*
3. *There exist no two jobs $j' \succ j$, such that a portion of j is processed after some portion of j', and before $d_{j'}$.*

Proof.

1. Since $FAST(\mathcal{J}') = \{j_i^d : 1 \leq i \leq \ell\}$ (by Proposition 1.2.2), and OPT is a YDS-extension, it follows that each j_i^d is processed exactly in its allowed interval I_i.
2. (a) Assume for the sake of contradiction that $[t_i, t_{i+1}) \subseteq \mathcal{T}$ contains a number of maximal intervals $N_1, N_2, \dots, N_\psi$ (ordered from left to right[2]) during which jobs are being processed, with $\psi \geq 2$. Let $M_1, M_2, \dots, M_{\psi'}$ (again ordered from left to right) be the remaining maximal intervals in $[t_i, t_{i+1})$, so that $N_1, \dots, N_\psi$ and $M_1, \dots, M_{\psi'}$ partition the zone $[t_i, t_{i+1})$. Furthermore, note that for each $i = 1, \dots, \psi'$, the processor is either active but idle or asleep during the whole interval M_i, since otherwise setting the processor asleep during the whole interval M_i would incur a strictly smaller energy consumption.

   We modify the schedule by shifting the intervals N_i, $i = 2, \dots, \psi$ to the left, so that $N_1, N_2, \dots, N_\psi$ now form a single contiguous processing interval. The

[2]For any two time points $t_1 < t_2$, we say that t_1 is to the *left* of t_2, and t_2 is to the *right* of t_1.

intervals M_k lying to the right of N_1 are moved further right and merge into a single (longer) interval M' during which no jobs are being processed. If the processor was active during each of these intervals M_k, then we keep the processor active during the new interval M', else we transition it to the sleep state. We observe that the resulting schedule is still a YDS-extension (note that its energy consumption is at most that of the initial schedule), but is lexicographically smaller.

For the second part of the statement, assume that there exists exactly one contiguous maximal processing interval N_1 within $[t_i, t_{i+1})$, and that there exist two M-intervals, M_1 and M_2 before and after N_1, respectively.

We consider two cases:

- The processor is active just before t_i, or the processor is asleep both just before t_i and just after t_{i+1}: In this case we can shift N_1 left by $|M_1|$ time units, so that it starts at t_i. Again, we keep the processor active during $[t_i + |N_1|, t_{i+1})$ only if it was active during both M_1 and M_2. As before, the resulting schedule remains a YDS-extension, and is lexicographically smaller.
- The processor is in the sleep state just before t_i but active just after t_{i+1}: In this case we shift N_1 by $|M_2|$ time units to the right, so that its right endpoint becomes t_{i+1}. During the new idle interval $[t_i, t_i + |M_1| + |M_2|)$ we set the processor asleep. Note that in this case the processor was asleep during M_1. The schedule remains a YDS-extension, but its energy consumption becomes strictly smaller: (i) either the processor was asleep during M_2, in which case the resulting schedule uses the same energy while the processor is active but has one wake-up operation less, or (ii) the processor was active and idle during M_2, in which case the resulting schedule saves the idle energy that was expended during M_2.

(b) The statement follows directly from the second property of Lemma 1.2.1 and the fact that all jobs processed during $[t_i, t_{i+1})$ belong to $SLOW(\mathcal{J}')$ and are active in the entire zone.

3. Assume for the sake of contradiction that there exist two jobs $j' \succ j$, such that a portion of j is processed during an interval $Z = [\zeta_1, \zeta_2)$, $\zeta_2 \leq d_{j'}$, and some portion of j' is processed during an interval $Z' = [\zeta_1', \zeta_2')$, with $\zeta_2' \leq \zeta_1$. We first observe that both jobs belong to $SLOW(\mathcal{J}')$. This follows from the fact that both jobs are active during the whole interval $[\zeta_1', \zeta_2)$, and processed during parts of this interval, whereas any job j_i^d (which are the only jobs in $FAST(\mathcal{J}')$) is processed exactly in its entire interval $[y_i, z_i)$ (by statement 1 of the lemma).

   By the second property of Lemma 1.2.1, both j and j' are processed at the same speed. We can now apply a swap argument. Let $L := \min\{|Z|, |Z'|\}$. Note that OPT schedules only j' during $[\zeta_2' - L, \zeta_2')$ and only j during $[\zeta_2 - L, \zeta_2)$. Swap the part of the schedule OPT in $[\zeta_2' - L, \zeta_2')$ with the schedule in the interval $[\zeta_2 - L, \zeta_2)$.

Given the above observations, it can be easily verified that the resulting schedule (i) is feasible and remains a YDS-extension, and (ii) is lexicographically smaller than OPT.

□

The next lemma shows how to transform the lexicographically minimal YDS-extension for $\mathcal{J}'$ of the previous lemma into a well-ordered discretized schedule. This is the most crucial part of our approach. Roughly speaking, the transformation needs to guarantee that (1) in each zone, the volume of a job $j \in SLOW(\mathcal{J}')$ processed is an integer multiple of $v_j/(4n^2\lceil 1/\delta \rceil)$ (this is tantamount to making sure that each zone has integral job pieces to deal with), (2) the job pieces start and end at the time points in W, and (3) all the job pieces are processed in the "right order". As we will show, the new schedule may run at a higher speed than the given lexicographically minimal YDS-extension, but not by too much.

Lemma 1.3.6. *Let OPT be a lexicographically minimal YDS-extension for $\mathcal{J}'$, and let $s_{\mathcal{S}}(t)$ denote the speed of schedule $\mathcal{S}$ at time t, for any $\mathcal{S}$ and t. Then there exists a well-ordered discretized schedule F, such that at any time point $t \in \mathcal{T}$, there holds*

$$s_F(t) \leq (1+\delta)^3 s_{OPT}(t),$$

and for every $t \notin \mathcal{T}$, there holds

$$s_F(t) = s_{OPT}(t).$$

Proof. Through a series of three transformations, we will transform OPT to a well-ordered discretized schedule F, while upper bounding the increase in speed caused by each of these transformations. More specifically, we will transform OPT to a schedule F_1 satisfying (i) and (iii) of Definition 1.3.3, then F_1 to F_2 where we slightly adapt the block lengths, and finally F_2 to F which satisfies all three properties of Definition 1.3.3. Each of these transformations can increase the speed by at most a factor $(1+\delta)$ for any $t \in \mathcal{T}$ and does not affect the speed in any interval I_i.

Transformation 1 (OPT $\rightarrow F_1$): We will transform the schedule so that

(i) For each job $j \in SLOW(\mathcal{J}')$, an integer multiple of $v_j/(4n^2\lceil 1/\delta \rceil)$ volume of job j is processed in each zone, and the processing order of jobs within each zone is determined by $\prec$. Together with property 1 of Lemma 1.3.5, this implies that F_1 (considered as a schedule for pieces) satisfies Definition 1.3.3(i).

(ii) The well-ordered property of Definition 1.3.3 is satisfied.

(iii) For all $t \in \mathcal{T}$ it holds that $s_{F_1}(t) \leq (1+\delta) s_{OPT}(t)$, and for every $t \notin \mathcal{T}$ it holds that $s_{F_1}(t) = s_{OPT}(t)$.

Note that by Lemma 1.3.5, every zone is either empty, filled exactly by a job j_i^d, or contains a single block. For any job $j \in SLOW(\mathcal{J}')$, and every zone $[t_i, t_{i+1})$, let

V_j^i be the processing volume of job j that OPT schedules in zone $[t_i, t_{i+1})$. Since there can be at most $2n$ different zones, for every job j there exists some index $h(j)$, such that $V_j^{h(j)} \geq v_j/(2n)$.

For every job $j \in SLOW(\mathcal{J}')$, and every $i \neq h(j)$, we reduce the load of job j processed in $[t_i, t_{i+1})$, by setting it to

$$\bar{\mathcal{V}}_j^i = \left\lfloor V_j^i / \frac{v_j}{4n^2\lceil 1/\delta \rceil)} \right\rfloor \cdot \frac{v_j}{4n^2\lceil 1/\delta \rceil}.$$

Finally, we set the volume of j processed in $[t_{h(j)}, t_{h(j)+1})$ to $\bar{\mathcal{V}}_j^{h(j)} = v_j - \sum_{i \neq h(j)} \bar{\mathcal{V}}_j^i$. To keep the schedule feasible, we process the new volume of each non-empty zone $[t_i, t_{i+1}) \subseteq \mathcal{T}$ in the zone's original block B_i, at a uniform speed of $\sum_{j \in SLOW(\mathcal{J}')} (\bar{\mathcal{V}}_j^i)/|B_i|$. Here, the processing order of the jobs within the block is determined by $\prec$.

Note that in the resulting schedule F_1, a job may be processed at different speeds in different zones, but each zone uses only one constant speed level.

It is easy to see that F_1 is a feasible schedule in which for each job $j \in SLOW(\mathcal{J}')$, an integer multiple of $v_j/(4n^2\lceil 1/\delta \rceil)$ volume of j is processed in each zone, and that $\bar{\mathcal{V}}_j^i \leq V_j^i$ for all $i \neq h(j)$. Furthermore, if $i = h(j)$, we have that $\bar{\mathcal{V}}_j^i - V_j^i \leq v_j/(2n\lceil 1/\delta \rceil)$, and $V_j^i \geq v_j/(2n)$. It follows that $\bar{\mathcal{V}}_j^i \leq V_j^i + V_j^i/\lceil 1/\delta \rceil \leq (1+\delta)V_j^i$ in this case, and therefore $s_{F_1}(t) \leq (1+\delta)s_{OPT}(t)$ for all $t \in \mathcal{T}$. We note here, that for every job j_i^d, and the corresponding interval I_i, nothing changes during the transformation.

We finally show that F_1 satisfies the well-ordered property of Definition 1.3.3. Assume for the sake of contradiction that there exists a piece u ending at some t, and there exists a piece $u' \succ u$ with $d_{u'} \geq t$ that is scheduled before t. Recall that we can implicitly assume that the pieces of any fixed job are processed in the corresponding order $\prec$. Therefore $job(u') \succ job(u)$, by definition of the ordering $\prec$ among pieces. Furthermore, if $[t_k, t_{k+1})$ and $[t_{k'}, t_{k'+1})$ are the zones in which u and u', respectively, are scheduled, then $k' < k$, as $k' = k$ would contradict F_1's processing order of jobs inside a zone. Also note that $d_{u'} \geq t_{k+1}$, since $t \in (t_k, t_{k+1}]$, and (t_k, t_{k+1}) does not contain any deadline. This contradicts property 3 of Lemma 1.3.5, as the original schedule OPT must have processed some volume of $job(u')$ in $[t_{k'}, t_{k'+1})$, and some volume of $job(u)$ in $[t_k, t_{k+1})$.

Transformation 2 ($F_1 \to F_2$): In this transformation, we slightly modify the block lengths, as a preparation for Transformation 3. For every non-empty zone $[t_i, t_{i+1}) \subseteq \mathcal{T}$, we increase the uniform speed of its block until it has a length of $(1+\delta)^j \frac{1}{4n^2 s_{crit}(1+\delta)\lceil 1/\delta \rceil}$ for some integer $j \geq 0$, keeping one of its endpoints fixed at t_i or t_{i+1}. Note that in F_1, the block had length at least $\frac{1}{4n^2 s_{crit}(1+\delta)\lceil 1/\delta \rceil}$, since it contained a volume of at least $1/(4n^2\lceil 1/\delta \rceil)$, and the speed in this zone was at most $(1+\delta)s_{crit}$. The speedup needed for this modification is clearly at most $(1+\delta)$.

As this transformation does not change the processing order of any pieces nor the zone in which any piece is scheduled, it preserves the well-ordered property of Definition 1.3.3.

Transformation 3 ($F_2 \to F$): In this final transformation, we want to establish Definition 1.3.3(ii). To this end, we shift and compress certain pieces in F_2, such that every execution interval starts and ends at a time point from W (this is already true for pieces corresponding to jobs j_i^d). The procedure resembles a transformation done in [46]. For any zone $[t_i, t_{i+1}) \subseteq \mathcal{T}$, we do the following: Consider the pieces that F_2 processes within the zone $[t_i, t_{i+1})$, and denote this set of pieces by D_i. If $D_i = \emptyset$, nothing needs to be done. Otherwise, let γ be the integer such that $(1+\delta)^\gamma \frac{1}{4n^2 s_{crit}(1+\delta)\lceil 1/\delta \rceil}$ is the length of the block in this zone, and let

$$\Delta := \frac{(1+\delta)^\gamma \frac{1}{4n^2 s_{crit}(1+\delta)\lceil 1/\delta \rceil}}{16n^6 \lceil 1/\delta \rceil^2 (1 + \lceil 1/\delta \rceil)}.$$

Note that in the definition of W, we introduced $16n^6 \lceil 1/\delta \rceil^2 (1 + \lceil 1/\delta \rceil)$ many time points (for $j = \gamma$ and $r = 1, \dots, 16n^6 \lceil 1/\delta \rceil^2 (1+\lceil 1/\delta \rceil)$) that subdivide this block into $16n^6 \lceil 1/\delta \rceil^2 (1 + \lceil 1/\delta \rceil)$ intervals of length Δ. Furthermore, since $|D_i| \leq 4n^3 \lceil 1/\delta \rceil$, there must exist a piece $u \in D_i$ with execution time $\Gamma_u \geq 4n^3 \lceil 1/\delta \rceil (1+\lceil 1/\delta \rceil)\Delta$. We now partition the pieces in $D_i \setminus u$ into D^+, the pieces processed after u, and D^-, the pieces processed before u. First, we restrict our attention to D^+. Let $q_1, \dots, q_{|D^+|}$ denote the pieces in D^+ in the order they are processed by F_2. Starting with the last piece $q_{|D^+|}$, and going down to q_1, we modify the schedule as follows. We keep the end of $q_{|D^+|}$'s execution interval fixed, and shift its start to the next earlier time point in W, reducing its uniform execution speed accordingly. At the same time, to not produce any overlappings, we shift the execution intervals of all $q_k,\ k < |D^+|$ by the same amount to the left (leaving their lengths unchanged). Eventually, we also move the execution end point of u by the same amount to the left (leaving its start point fixed). This shortens the execution interval of u and "absorbs" the shifting of the pieces in D^+ (note that the processing speed of u increases as its interval gets shorter). We then proceed with $q_{|D^+|-1}$, keeping its end (which now already resides at a time point in W) fixed, and moving its start to the next earlier time point in W. Again, the shift propagates to earlier pieces in D^+, which are moved by the same amount, and shortens u's execution interval once more. When all pieces in D^+ have been modified in this way, we turn to D^- and apply the same procedure there. This time, we keep the start times fixed and instead shift the right end points of the execution intervals further to the right. As before, u "absorbs" the propagated shifts, as we increase its start time accordingly. After this modification, the execution intervals of all pieces in D_i start and end at time points in W.

To complete the proof, we need to argue that the speedup of piece u is bounded by a factor $(1+\delta)$. Since $|D_i| \leq 4n^3 \lceil 1/\delta \rceil$, u's execution interval can be shortened at most $4n^3 \lceil 1/\delta \rceil$ times, each time by a length of at most Δ. Furthermore, recall that the execution time of u was $\Gamma_u \geq 4n^3 \lceil 1/\delta \rceil (1 + \lceil 1/\delta \rceil)\Delta$. Therefore, its new execution time is at least $\Gamma_u - 4n^3 \lceil 1/\delta \rceil \Delta \geq \Gamma_u - \frac{\Gamma_u}{1+\lceil 1/\delta \rceil}$, and the speedup factor thus at most

$$\frac{\Gamma_u}{\Gamma_u - \frac{\Gamma_u}{1+\lceil 1/\delta \rceil}} = \frac{1}{1 - \frac{1}{1+\lceil 1/\delta \rceil}} \leq 1 + \delta.$$

Again, the transformation does not change the processing order of any pieces nor the zone in which any piece is scheduled, and thus preserves the well-ordered property of Definition 1.3.3. □

We now show that the speedup used in our transformation does not increase the energy consumption by more than a factor of $1+\epsilon$. To this end, observe that for any $t \in \mathcal{T}$, the speed of the schedule OPT in Lemma 1.3.6 is at most s_{crit}, by Lemma 1.3.5(2). Furthermore, note that the final schedule F has speed zero whenever OPT has speed zero. This allows F to use exactly the same sleep phases as OPT (resulting in the same wake-up costs). It therefore suffices to prove the following lemma, in order to bound the increase in energy consumption.

Lemma 1.3.7. *For any $s \in [0, s_{crit}]$, there holds*

$$\frac{P\big((1+\delta)^3 s\big)}{P(s)} \leq 1+\epsilon.$$

Proof.

$$\begin{aligned}
\frac{P\big((1+\delta)^3 s\big)}{P(s)} &\overset{(1)}{\leq} \frac{P\big((1+4\delta)s\big)}{P(s)} \\
&= \frac{P(s) + 4\delta s \frac{P(s+4\delta s)-P(s)}{4\delta s}}{P(s)} \\
&\overset{(2)}{\leq} \frac{P(s) + 4\delta s \frac{P(s+s_{crit})-P(s)}{s_{crit}}}{P(s)} \\
&\overset{(3)}{\leq} \frac{P(s) + 4\delta s \frac{P(2s_{crit})-P(s_{crit})}{s_{crit}}}{P(s)} \\
&\overset{(4)}{\leq} 1 + 4\delta \frac{s_{crit}}{P(s_{crit})} \cdot \frac{P(2s_{crit}) - P(s_{crit})}{s_{crit}} \\
&\overset{(5)}{\leq} 1+\epsilon.
\end{aligned}$$

In the above chain of inequalities, (1) holds since $\delta \leq \frac{1}{4}$ and $P(s)$ is non-decreasing. (2) and (3) follow from the convexity of $P(s)$, and the fact that $4\delta s \leq s_{crit}$. Inequality (4) holds since s_{crit} minimizes $P(s)/s$ (and thus maximizes $s/P(s)$), and (5) follows from the definition of δ. □

We summarize the major result of this section in the following lemma.

Lemma 1.3.8. *There exists a well-ordered discretized schedule with an energy consumption no more than $(1+\epsilon)$ times the optimal energy consumption for $\mathcal{J}'$.*

1.4. The Dynamic Program

In this section, we show how to use dynamic programming to find a well-ordered discretized schedule with minimum energy consumption. In the following, we discuss only how to find the minimum energy consumption of this target schedule, as the actual schedule can be easily retrieved by proper bookkeeping in the dynamic programming process.

Recall that D is the set of all pieces and W the set of time points. Let $u_1, u_2, \ldots, u_{|D|}$ be the pieces in D, and w.l.o.g. assume that $u_1 \prec u_2 \prec \ldots \prec u_{|D|}$.

Definition 1.4.1. *For any $k \in \{1, \ldots, |D|\}$, and $\tau_1 \leq \tau_2$, τ_1, $\tau_2 \in W$, we define $E_k(\tau_1, \tau_2)$ as the minimum energy consumption during the interval $[\tau_1, \tau_2]$, of a well-ordered discretized schedule so that*

1. *all pieces $\{u \succeq u_k : \tau_1 < d_u \leq \tau_2\}$ are processed in the interval $[\tau_1, \tau_2)$, and*
2. *the machine is active right before τ_1 and right after τ_2.*

In case that there is no such feasible schedule, let $E_k(\tau_1, \tau_2) = \infty$.

The DP proceeds by filling the entries $E_k(\tau_1, \tau_2)$ by decreasing index of k. The base cases are

$$E_{|D|+1}(\tau_1, \tau_2) := \min\{P(0)(\tau_2 - \tau_1), C\},$$

for all $\tau_1, \tau_2 \in W, \tau_1 \leq \tau_2$. For the recursion step, suppose that we are about to fill in $E_k(\tau_1, \tau_2)$. There are two possibilities.

- Suppose that $d_{u_k} \notin (\tau_1, \tau_2]$. Then clearly $E_k(\tau_1, \tau_2) = E_{k+1}(\tau_1, \tau_2)$.
- Suppose that $d_{u_k} \in (\tau_1, \tau_2]$. By definition, piece u_k needs to be processed in the interval $[\tau_1, \tau_2)$. We need to guess its actual execution period $[b, e) \subseteq [\tau_1, \tau_2)$, and process the remaining pieces $\{u \succeq u_{k+1} : \tau_1 < d_u \leq \tau_2\}$ in the two intervals $[\tau_1, b)$ and $[e, \tau_2)$. We first rule out some guesses of $[b, e)$ that are bound to be wrong.
  - By Definition 1.3.3(i), in a discretized schedule, a piece has to be processed completely inside a zone $[t_i, t_{i+1})$ (recall that $t_i \in W'$ are the release times and deadlines of the jobs). Therefore, in the right guess, the interior of $[b, e)$ does not contain any release times or deadlines; more precisely, there is no time point $t_i \in W'$ so that $b < t_i < e$.
  - By Definition 1.3.3(iii), in a well-ordered discretized schedule, if piece u_k ends at time point e, then all pieces $u' \succ u_k$ with deadline $d_{u'} \geq e$ are processed *after* u_k. However, consider the guess $[b, e)$, where $e = d_{u'}$ for some $u' \succ u_k$ (notice that the previous case does not rule out this possibility). Then u' cannot be processed anywhere in a well-ordered schedule. Thus, such a guess $[b, e)$ cannot be right.

By the preceding discussion, if the guess (b, e) is right, the two sets of pieces $\{u \succeq u_{k+1} : \tau_1 < d_u \leq b\}$ and $\{u \succeq u_{k+1} : e < d_u \leq \tau_2\}$, along with piece u_k, comprise all pieces to be processed that are required by the definition of $E_k(\tau_1, \tau_2)$. Clearly, the former set of pieces $\{u \succeq u_{k+1} : \tau_1 < d_u \leq b\}$ has to be processed in the interval $[\tau_1, b)$; the latter set of pieces, in a well-ordered schedule, must be processed in the interval $[e, \tau_2)$ if $[b, e)$ is the correct guess for the execution of the piece u_k.

We therefore have that

$$E_k(\tau_1, \tau_2) = \min_{\substack{b,e \in W,\ [b,e)\subseteq[\tau_1,\tau_2), \\ [b,e)\subseteq[r_{u_k}, d_{u_k}), \\ \nexists t_i \in W', \text{ s.t. } b<t_i<e, \\ \nexists u' \succ u_k, \text{ s.t. } d_{u'}=e.}} \Big\{ E_{k+1}(\tau_1, b) + P(\frac{v_{u_k}}{e-b})(e-b) + E_{k+1}(e, \tau_2) \Big\}$$

if there exist $b, e \in W$ with the properties stated under the min-operator, and $E_k(\tau_1, \tau_2) = \infty$ otherwise.

It can be verified that the running time of the DP is polynomial in the input size and $1/\epsilon$. The minimum energy consumption for the target schedule is $E_1(0, d_{max})$.

Theorem 1.4.2. *There exists a fully polynomial-time approximation scheme (FPTAS) for speed scaling with sleep state.*

Proof. Given an arbitrary instance $\mathcal{J}$ for speed scaling with sleep state, we can transform it in polynomial time to an instance $\mathcal{J}'$, as seen in Section 1.2. We then apply the dynamic programming algorithm that was described in this section to obtain a well-ordered discretized schedule $\mathcal{S}'$ of minimal energy consumption for instance $\mathcal{J}'$. By Lemma 1.3.8, we have that $\mathcal{S}'$ is a $(1+\epsilon)$-approximation for instance $\mathcal{J}'$. Furthermore, note that every discretized schedule (and therefore also $\mathcal{S}'$) executes each job j_i^d exactly in its allowed interval $I_i = [y_i, z_i)$. This holds because there are no time points from the interior of I_i included in W, and any discretized schedule must therefore choose to run j_i^d precisely from $y_i \in W$ to $z_i \in W$. Therefore, by Lemma 1.2.3, we can transform $\mathcal{S}'$ to a schedule $\mathcal{S}$ in polynomial time and obtain a $(1+\epsilon)$-approximation for $\mathcal{J}$. ☐

2. Non-Preemptive Speed Scaling

2.1. Introduction

The first theoretical model for speed scaling problems was introduced by Yao et al. in their seminal paper of 1995 [79]. One is given a set of jobs, each with its own *volume* v_j (number of CPU cycles needed for completion of this job), *release time* r_j (when the job becomes available), and *deadline* d_j (when the job needs to be finished), and a processor with power function $P(s) = s^\alpha$, where s is the processing speed, and $\alpha > 1$ is a constant (typically between two and three for modern microprocessors [30, 77]). The energy consumption is power integrated over time, and the objective is to process all given jobs within their *time windows* $[r_j, d_j)$, while minimizing the total energy consumption.

Most work in the literature focuses on the *preemptive* version of the problem, where the execution of a job may be interrupted and resumed at a later point of time. For this setting, Yao et al. [79] gave a polynomial-time exact algorithm to compute the optimal schedule. The *non-preemptive* model, where a job must be processed uninterruptedly until its completion, has so far received surprisingly little attention, even though it is often preferred in practice and widely used in current real-life applications. For example, most current real-time operating systems for automotive applications use non-preemptive scheduling as defined by the OSEK/VDX standard [67]. The advantage of this strategy lies in the significant lower overhead (preemption requires to memorize and restore the state of the system and the job) [8], and the avoidance of synchronization efforts for shared resources [67]. From a theoretical point of view, the non-preemptive model is of interest, since it is a natural variation of Yao et al.'s original model. So far, little is known about the complexity of the non-preemptive speed scaling problem. On the negative side, no lower bound is known, except that the problem is strongly NP-hard [8]. On the positive side, Antoniadis and Huang [8] showed that the problem has a constant factor approximation algorithm, although the obtained factor $2^{5\alpha-4}$ is rather large. However, recently a number of papers have appeared that (significantly) improve upon the constant [16, 17, 34].

2.1.1. Our Results and Techniques

We work towards better understanding the complexity of the non-preemptive speed scaling problem, by considering several special cases and presenting (near-)optimal algorithms. In the following, we give a summary of our results.

Laminar Instances: An instance is said to be *laminar* if for any two different jobs j_1 and j_2, either $[r_{j_1}, d_{j_1}) \subseteq [r_{j_2}, d_{j_2})$, or $[r_{j_2}, d_{j_2}) \subseteq [r_{j_1}, d_{j_1})$, or $[r_{j_1}, d_{j_1}) \cap [r_{j_2}, d_{j_2}) = \emptyset$. The problem remains strongly NP-hard for this case [8]. We present the first $(1+\epsilon)$-approximation for this problem, with a quasipolynomial running time (i.e. a running time bounded by $2^{poly(\log n)}$ for any fixed $\epsilon > 0$); a so-called quasipolynomial-time approximation scheme (QPTAS). Our result implies that laminar instances are not APX-hard, unless NP $\subseteq$ DTIME($2^{poly(\log n)}$). We remark that laminar instances form an important subclass of instances that not only arise commonly in practice (e.g. when jobs are created by recursive function calls [62]), but are also of theoretical interest, as they highlight the difficulty of the non-preemptive speed scaling problem: Taking instances with an "opposite" structure, namely *agreeable instances* (here for any two jobs j_1 and j_2 with $r_{j_1} < r_{j_2}$, it holds that $d_{j_1} < d_{j_2}$), the problem becomes polynomial-time solvable [8]. On the other hand, further restricting the instances from laminar to *purely-laminar* (see next case) results in a problem that is only weakly NP-hard and admits an FPTAS.

Purely-Laminar Instances: An instance is said to be *purely-laminar* if for any two different jobs j_1 and j_2, either $[r_{j_1}, d_{j_1}) \subseteq [r_{j_2}, d_{j_2})$, or $[r_{j_2}, d_{j_2}) \subseteq [r_{j_1}, d_{j_1})$. We present a fully polynomial-time approximation scheme (FPTAS) for this class of instances. This is the best possible result (unless P = NP), as the problem is still (weakly) NP-hard [8].

Equal-Volume Jobs: If all jobs have the same volume $v_1 = v_2 = \ldots = v_n = v$, we present a polynomial-time algorithm for computing an (exact) optimal schedule. We thereby improve upon a result of Bampis et al. [15], who proposed a 2^α-approximation algorithm, and answer their question for the complexity status of this problem.

Bounded Number of Time Windows: If the total number of different time windows is bounded by a constant, we present an FPTAS for the problem. This result is again optimal (unless P = NP), as the problem remains (weakly) NP-hard even if there are only two different time windows [8].

The basis of all our results is a discretization of the problem, in which we allow the processing of any job to start and end only at a carefully chosen set of *grid points* on the time axis. We then use dynamic programming to solve the discretized problem. For laminar instances, however, even computing the optimal discretized solution is hard. The main technical contribution of our QPTAS is a relaxation that decreases the exponential size of the DP-tableau without adding too much energy cost. For this, we use an overly compressed representation of job sets in the bookkeeping. Roughly speaking, we "lose" a number of jobs in each step of the recursion, but we ensure that these jobs can later be scheduled with only a small increment of energy cost.

2.1.2. Related Work

The study of dynamic speed scaling problems for reduced energy consumption was initiated by Yao, Demers, and Shenker in 1995. In their seminal paper [79], they presented a polynomial-time algorithm for finding an optimal schedule when preemption of jobs is allowed. Furthermore, they also studied the online version of the problem (again with preemption of jobs allowed), where jobs become known only at their release times, and developed two constant-competitive algorithms called *Average Rate* and *Optimal Available*.

Over the years, a rich spectrum of variations and generalizations of the original model have been investigated, mostly with a focus on the preemptive version. Irani et al. [49], for instance, considered a setting where the processor additionally has a sleep state available. Another extension of the original model is to restrict the set of possible speeds that we may choose from, for example by allowing only a number of discrete speed levels [33, 61], or bounding the maximum possible speed [20, 32, 45]. Variations with respect to the objective function have also been studied, for instance by Albers and Fujiwara [3] and Bansal et al. [19], who tried to minimize a combination of energy consumption and total flow time of the jobs. Finally, the problem has also been studied for arbitrary power functions [21], as well as for multiprocessor settings [5, 6, 28].

In contrast to this diversity of results, the non-preemptive version of the speed scaling problem has been addressed rarely in the literature. Only in 2012, Antoniadis and Huang [8] proved that the problem is strongly NP-hard, and gave a $2^{5\alpha-4}$-approximation algorithm for the general case. This ratio has then been improved to $2^{\alpha-1}(1+\epsilon)\tilde{B}_\alpha$ by Bampis et al. [16], where $\tilde{B}_\alpha$ is a generalization of the Bell number to fractional α-values, and to $(12(1+\epsilon))^{\alpha-1}$ by Cohen-Addad et al. [34]. Later, Bampis et al. [17] improved the ratio further to $(1+\epsilon)\tilde{B}_\alpha$. For the special case where all jobs have the same volume, Bampis et al. [15] proposed a 2^α-approximation algorithm. Independently of our result for this setting, Angel et al. [7] also gave a polynomial-time exact algorithm for such instances.

Multi-processor non-preemptive speed scaling also started to draw the attention of researchers. See [15, 34] for details.

2.1.3. Overview

In Section 2.2 we give a formal definition of the problem and establish a couple of preliminaries. We then present our QPTAS for laminar instances (Section 2.3), and our polynomial-time algorithm for instances with equal-volume jobs (Section 2.4). Our FPTAS' for purely-laminar instances and instances with a bounded number of different time windows follow in Sections 2.5 and 2.6, respectively.

2.2. Preliminaries and Notations

The input is given by a set $\mathcal{J}$ of n jobs, each having its own release time r_j, deadline d_j, and volume $v_j > 0$. The power function of the speed-scalable processor is $P(s) = s^\alpha$, with $\alpha > 1$, and the energy consumption is power integrated over time. A *schedule* specifies for any point of time (i) which job to process, and (ii) which speed to use. A schedule is called *feasible* if every job is executed entirely within its time window $[r_j, d_j)$, which we will also call the *allowed interval* of job j. Preemption is not allowed, meaning that once a job is started, it must be executed entirely until its completion. Our goal is to find a feasible schedule of minimum total energy consumption.

We use $E(S)$ to denote the total energy consumed by a given schedule S, and $E(S, j)$ to denote the energy used for the processing of job j in schedule S. Furthermore, we use OPT to denote the energy consumption of an optimal schedule. A crucial observation is that, due to the convexity of the power function $P(s) = s^\alpha$, it is never beneficial to vary the speed during the execution of a job. This follows from Jensen's Inequality. We can therefore assume that in an optimal schedule, every job is processed using a uniform speed.

In the following, we restate a proposition from [8], which allows us to speed up certain jobs without paying too much additional energy cost.

Proposition 2.2.1. *Let S and S' be two feasible schedules that process j using uniform speeds s and $s' > s$, respectively. Then $E(S', j) = (s'/s)^{\alpha-1} \cdot E(S, j)$.*

Proof.

$$E(S', j) = P(s')\frac{v_j}{s'} = (s')^{\alpha-1} v_j = (\frac{s'}{s})^{\alpha-1} s^{\alpha-1} v_j$$
$$= (\frac{s'}{s})^{\alpha-1} P(s)\frac{v_j}{s} = (\frac{s'}{s})^{\alpha-1} E(S, j).$$

□

As mentioned earlier, all our results rely on a discretization of the time axis, in which we focus only on a carefully chosen set of time points. We call these points *grid points* and define *grid point schedules* as follows.

Definition 2.2.2 (Grid Point Schedule)**.** *A schedule is called* grid point schedule *if the processing of every job starts and ends at a grid point.*

We use two different sets of grid points, $\mathcal{P}_{\text{approx}}$ and $\mathcal{P}_{\text{exact}}$. The first set, $\mathcal{P}_{\text{approx}}$, is more universal, as it guarantees the existence of a near-optimal grid point schedule for any kind of instances. On the contrary, the set $\mathcal{P}_{\text{exact}}$ is specialized for the case of equal-volume jobs, and on such instances guarantees the existence of a grid point schedule with energy consumption exactly OPT. We now give a detailed description of both sets. For this, let us call a time point t an *event* if $t = r_j$ or $t = d_j$ for some job j, and let $t_1 < t_2 < \ldots < t_p$ be the set of ordered events. We call the interval

between two consecutive events t_i and t_{i+1} a *zone*. Furthermore, let $\gamma := 1 + \lceil 1/\epsilon \rceil$, where $\epsilon > 0$ is the error parameter of our approximation schemes.

Definition 2.2.3 (Grid Point Set $\mathcal{P}_{\text{approx}}$)**.** *The set $\mathcal{P}_{\text{approx}}$ is obtained in the following way. First, create a grid point at every event. Secondly, for every zone (t_i, t_{i+1}), create $n^2\gamma - 1$ equally spaced grid points that partition the zone into $n^2\gamma$ many subintervals of equal length $L_i = \frac{t_{i+1}-t_i}{n^2\gamma}$. Now $\mathcal{P}_{\text{approx}}$ is simply the union of all created grid points.*

Note that the total number of grid points in $\mathcal{P}_{\text{approx}}$ is at most $\mathcal{O}\big(n^3(1 + \frac{1}{\epsilon})\big)$, as there are $\mathcal{O}(n)$ zones, for each of which we create $\mathcal{O}(n^2\gamma)$ grid points.

Lemma 2.2.4. *There exists a grid point schedule $\mathcal{G}$ with respect to $\mathcal{P}_{\text{approx}}$, such that $E(\mathcal{G}) \leq (1+\epsilon)^{\alpha-1}\text{OPT}$.*

Proof. Let $\mathcal{S}^*$ be an optimal schedule, that is $E(\mathcal{S}^*) = \text{OPT}$. We show how to modify $\mathcal{S}^*$ by shifting and compressing certain jobs, s.t. every execution interval starts and ends at a grid point. For the proof we focus on one particular zone (t_i, t_{i+1}), and the lemma follows by applying the transformation to each other zone individually.

Let us consider the jobs that $\mathcal{S}^*$ processes within the zone (t_i, t_{i+1}). If a job's execution interval overlaps partially with this zone, we consider only its fraction inside (t_i, t_{i+1}) and treat this fraction as if it were a job by itself. We denote the set of (complete and partial) jobs in zone (t_i, t_{i+1}) by J. If $J = \emptyset$, nothing needs to be done. Otherwise, we can assume that $\mathcal{S}^*$ uses the entire zone (t_i, t_{i+1}) without any idle periods to process the jobs in J. If this were not the case, we could slow down the processing of any job in J without violating a release time or deadline constraint, and thus obtain a feasible schedule with lower energy cost than $\mathcal{S}^*$, a contradiction. Consequently, the total time for processing J in $\mathcal{S}^*$ is $\gamma n^2 L_i$ (recall that $L_i = \frac{t_{i+1}-t_i}{n^2\gamma}$), and as $|J| \leq n$, there must exist a job $j \in J$ with execution time $T_j \geq \gamma n L_i$.

We now partition the jobs in $J \setminus j$ into J^+, the jobs processed after j, and J^-, the jobs processed before j. First, we restrict our attention to J^+. Let $q_1, \ldots, q_{|J^+|}$ denote the jobs in J^+ in the order they are processed by $\mathcal{S}^*$. Starting with the last job $q_{|J^+|}$, and going down to q_1, we modify the schedule as follows. We keep the end of $q_{|J^+|}$'s execution interval fixed, and shift its start to the next earlier grid point, reducing its uniform execution speed accordingly. At the same time, to not produce any overlappings, we shift the execution intervals of all $q_k,\ k < |J^+|$ by the same amount, in the direction of earlier times (leaving their lengths unchanged). Eventually, we also move the execution end point of j by the same amount towards earlier times (leaving its start point fixed). This shortens the execution interval of j and "absorbs" the shifting of the jobs in J^+. The shortening of j's execution interval is compensated by an appropriate increase of speed. We then proceed with $q_{|J^+|-1}$, keeping its end (which now already resides at a grid point) fixed, and moving its start to the next earlier grid point. Again, the shift propagates to earlier jobs in

J^+, which are moved by the same amount, and shortens j's execution interval once more. When all jobs in J^+ have been modified in this way, we turn to J^- and apply the same procedure there. This time, we keep the start times fixed and instead shift the right end points of the execution intervals towards later times. As before, j "absorbs" the propagated shifts, as we increase its start time accordingly. After this modification, the execution intervals of all jobs in J start and end at grid points only.

To complete the proof, we need to analyze the changes made in terms of energy consumption. Let $\mathcal{G}$ denote the schedule obtained by the above modification of $\mathcal{S}^*$. Obviously, for all $j' \in J \setminus j$, we have that $E(\mathcal{G}, j') \leq E(\mathcal{S}^*, j')$, as the execution intervals of those jobs are only prolonged during the transformation process, resulting in a less or equal execution speed. The only job whose processing time is possibly shortened, is j. Since $|J| \leq n$, it can be shortened at most n times, each time by a length of at most L_i. Remember that the execution time of j in $\mathcal{S}^*$ was $T_j \geq \gamma n L_i$. Therefore, in $\mathcal{G}$, its execution time is at least $T_j - nL_i \geq T_j - T_j/\gamma$. Thus the speedup factor of j in $\mathcal{G}$ compared to $\mathcal{S}^*$ is at most

$$\frac{T_j}{T_j - \frac{T_j}{\gamma}} = \frac{1}{1 - \frac{1}{\gamma}} \leq 1 + \epsilon,$$

where the last inequality follows from the definition of γ. Hence, Proposition 2.2.1 implies that $E(\mathcal{G}, j) \leq (1+\epsilon)^{\alpha-1} E(\mathcal{S}^*, j)$, and the lemma follows by summing up the energy consumptions of the individual jobs. □

Definition 2.2.5 (Grid Point Set $\mathcal{P}_{\text{exact}}$)**.** *For every pair of events $t_i \leq t_j$, and for every $k \in \{1, \ldots, n\}$, create $k-1$ equally spaced grid points that partition the interval $[t_i, t_j]$ into k subintervals of equal length. Furthermore, create a grid point at every event. The union of all these grid points defines the set $\mathcal{P}_{\text{exact}}$.*

Clearly, the total number of grid points in $\mathcal{P}_{\text{exact}}$ is $\mathcal{O}(n^4)$.

Lemma 2.2.6. *If all jobs have the same volume $v_1 = v_2 = \ldots = v_n = v$, there exists a grid point schedule $\mathcal{G}$ with respect to $\mathcal{P}_{\text{exact}}$, such that $E(\mathcal{G}) = \text{OPT}$.*

Proof. Let $\mathcal{S}^*$ be an optimal schedule. W.l.o.g., we can assume that $\mathcal{S}^*$ changes the processing speed only at events (recall that an event is either a release time or a deadline of some job), as a constant average speed between any two consecutive events minimizes the energy consumption (this follows from Jensen's Inequality) without violating release time or deadline constraints. Given this property, we will show that $\mathcal{S}^*$ is in fact a grid point schedule with respect to $\mathcal{P}_{\text{exact}}$. To this end, we partition the time horizon of $\mathcal{S}^*$ into *phases* of constant speed, that is time intervals of maximal length during which the processing speed is unchanged. As every job itself is processed using a uniform speed, no job is processed only partially within a phase. Each phase is therefore characterized by a pair of events $t_i \leq t_j$ indicating its beginning and end, and a number x of jobs that are processed completely between t_i and t_j at constant speed. It is clear that the grid points created for the pair (t_i, t_j)

and $k := x$ in the definition of $\mathcal{P}_{\text{exact}}$ correspond exactly to the start and end times of the jobs in this phase. Since this is true for every phase, $\mathcal{S}^*$ is indeed a grid point schedule. □

2.3. Laminar Instances

In this section, we present a QPTAS for laminar problem instances. We start with a small example to motivate our approach, in which we reuse some ideas of Muratore et al. [66] for a different scheduling problem. Consider Figure 2.1, where we have drawn a number of (laminar) time intervals, purposely arranged in a tree structure. Imagine that for each of those intervals I_k, we are given a set of jobs J_k whose allowed interval is equal to I_k. Furthermore, let us make the simplifying assumption that no job can "cross" the boundary of any interval I_k during its execution. Then, in any feasible schedule, the set of jobs J_1 at the root of the tree decomposes into two subsets; the set of jobs processed in the left child I_2, and the set of jobs processed in the right child I_3. Having a recursive procedure in mind, we can think of the jobs in

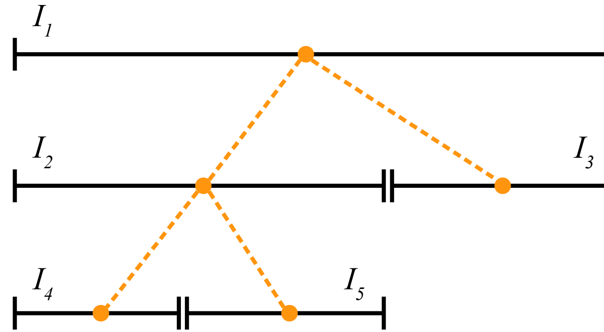


Figure 2.1.: Time intervals of a laminar instance, arranged in a tree structure.

the root as being split up and handed down to the respective children. Each child then has a set of "inherited" jobs, plus its own original jobs to process, and both are available throughout its whole interval. Now, the children also split up their jobs, and hand them down to the next level of the tree. This process continues until we finally reach the leaves of the tree, where we can simply execute the given jobs at a uniform speed over the whole interval.

Aiming for a reduced running time, we reverse the described process and instead compute the schedules in a bottom-up manner via dynamic programming, enumerating all possible sets of jobs that a particular node could "inherit" from its ancestors. This dynamic programming approach is the core part of our QPTAS, though it bears two major technical difficulties. The first one is that a job from a father node could also be scheduled "between" its children, starting in the interval of child one, stretching over its boundary, and entering the interval of child two. We overcome this issue by taking care of such jobs separately, and additionally listing the trun-

cated child-intervals in the dynamic programming tableau. The second (and main) difficulty is the huge number of possible job sets that a child node could receive from its parent. Reducing this number requires a controlled "omitting" of small jobs during the recursion, and a condensed representation of job sets in the DP tableau. At any point of time, we ensure that "omitted" jobs only cause a small increment of energy cost when being added to the final schedule. We now elaborate the details, beginning with a rounding of the job volumes. Let $\mathcal{I}$ be the original problem instance.

Definition 2.3.1 (Rounded Instance)**.** *The* rounded instance $\mathcal{I}'$ *is obtained by rounding down every job volume v_j to the next smaller number of the form $v_{\min}(1+\epsilon)^i$, where $i \in \mathbb{N}_{\geq 0}$ and $v_{\min}$ is the smallest volume of any job in the original instance. The numbers $v_{\min}(1+\epsilon)^i$ are called* size classes*, and a job* belongs *to size class $\mathcal{C}_i$ if its rounded volume is $v_{\min}(1+\epsilon)^i$.*

Lemma 2.3.2. *Every feasible schedule S' for $\mathcal{I}'$ can be transformed into a feasible schedule S for $\mathcal{I}$ with $E(S) \leq (1+\epsilon)^\alpha E(S')$.*

Proof. The lemma easily follows by using the same execution intervals as S' and speeding up accordingly. As rounded and original volume of a job differ by at most a factor of $1+\epsilon$, we need to increase the speed at any time t by at most this factor. Therefore the energy consumption grows by at most a factor of $(1+\epsilon)^\alpha$. □

From now on, we restrict our attention to the rounded instance $\mathcal{I}'$. Remember that our approach uses the inherent tree structure of the time windows. We proceed by formally defining a tree T that reflects this structure.

Definition 2.3.3 (Tree T)**.** *For every interval $[t_i, t_{i+1})$ between two consecutive events t_i and t_{i+1}, we introduce a vertex v. Additionally, we introduce a vertex for every time window $[r_j, d_j)$, $j \in \mathcal{J}$ that is not represented by a vertex yet. If several jobs share the same allowed interval, we add only one single vertex for this interval. The interval corresponding to a vertex v is denoted by I_v. We also associate a (possibly empty) set of jobs J_v with each vertex v, namely the set of jobs j whose allowed interval $[r_j, d_j)$ is equal to I_v. Finally, we specify a distinguished root node r as follows. If there exists a vertex v with $I_v = [r^*, d^*)$, where r^* is the earliest release time and d^* the latest deadline of any job in $\mathcal{J}$, we set $r := v$. Otherwise, we introduce a new vertex r with $I_r := [r^*, d^*)$ and $J_r := \emptyset$. The edges of the tree are defined in the following way. A node u is the son of a node v if and only if $I_u \subset I_v$ and there is no other node w with $I_u \subset I_w \subset I_v$. As a last step, we convert T into a binary tree by repeating the following procedure as long as there exists a vertex v with more than two children: Let v_1 and v_2 be two "neighboring" sons of v, such that $I_{v_1} \cup I_{v_2}$ forms a contiguous interval. Now create a new vertex u with $I_u := I_{v_1} \cup I_{v_2}$ and $J_u := \emptyset$, and make u a new child of v, and the new parent of v_1 and v_2. This procedure eventually results in a binary tree T with $\mathcal{O}(n)$ vertices.*

The main idea of our dynamic program is to stepwise compute schedules for subtrees of T, that is for the jobs associated with the vertices in the subtree (including

its root), plus a given set of "inherited" jobs from its ancestors. Enumerating all possible sets of "inherited" jobs, however, would burst the limits of our DP tableau. Instead, we use a condensed representation of those sets via so-called *job vectors*, focusing only on a logarithmic number of size classes and ignoring jobs that are too small to be covered by any of these. To this end, let δ be the smallest integer such that $n/\epsilon \leq (1+\epsilon)^\delta$, and note that δ is $\mathcal{O}(\log n)$ for any fixed $\epsilon > 0$.

Definition 2.3.4 (Job Vector)**.** *A* job vector $\overrightarrow{\lambda}$ *is a vector of* $\delta + 1$ *integers* $\lambda_0, \ldots, \lambda_\delta$*. The first component* λ_0 *specifies a size class, namely the largest out of* δ *size classes from which we want to represent jobs (therefore* $\lambda_0 \geq \delta - 1$*). The remaining* δ *components take values between* 0 *and* n *each, and define a number of jobs for each of the size classes* $\mathcal{C}_{\lambda_0}$, $\mathcal{C}_{\lambda_0 - 1}$, $\ldots$, $\mathcal{C}_{\lambda_0 - \delta + 1}$ *in this order. For example, if* $\delta = 2$*, the job vector* $(4, 2, 7)$ *defines a set containing* 2 *jobs with volume* $v_{\min}(1+\epsilon)^4$ *and* 7 *jobs with volume* $v_{\min}(1+\epsilon)^3$*.*
We refer to the set of jobs defined by a job vector $\overrightarrow{\lambda}$ *as* $J(\overrightarrow{\lambda})$*.*

Remark: We do not associate a strict mapping from the jobs defined by a job vector $\overrightarrow{\lambda}$ to the real jobs (given as input) they represent. The jobs $J(\overrightarrow{\lambda})$ should rather be seen as dummies that are used to reserve space and can be replaced by any real job of the same volume.

Definition 2.3.5 (Heritable Job Vector)**.** *A job vector* $\overrightarrow{\lambda} = (\lambda_0, \ldots, \lambda_\delta)$ *is* heritable *to a vertex* v *of* T *if:*

1. *At least* λ_i *jobs in* $\bigcup_{u \text{ ancestor of } v} J_u$ *belong to size class* $\mathcal{C}_{\lambda_0 - i + 1}$*, for* $1 \leq i \leq \delta$*.*

2. $\lambda_1 > 0$ *or* $\lambda_0 = \delta - 1$*.*

The conditions on a heritable job vector ensure that for a fixed vertex v, λ_0 can take only $\mathcal{O}(n)$ different values, as it must specify a size class that really occurs in the rounded instance, or be equal to $\delta - 1$. Therefore, in total, we can have at most $\mathcal{O}(n^{\delta+1})$ different job vectors that are heritable to a fixed vertex of the tree. In order to control the error caused by the laxity of our job set representation, we introduce the concept of δ-omitted schedules.

Definition 2.3.6 (δ-omitted Schedule)**.** *Let* J *be a given set of jobs. A* δ-omitted schedule *for* J *is a feasible schedule for a subset* $R \subseteq J$*, s.t. for every job* $j \in J \setminus R$*, there exists a job* big$(j) \in R$ *with volume at least* $v_j(1+\epsilon)^\delta$ *that is scheduled entirely inside the allowed interval of* j*. The jobs in* $J \setminus R$ *are called* omitted jobs*, the ones in* R non-omitted jobs*.*

Lemma 2.3.7. *Every* δ*-omitted schedule* S' *for a set of jobs* J *can be transformed into a feasible schedule* S *for all jobs in* J*, such that* $E(S) \leq (1+\epsilon)^\alpha E(S')$*.*

Proof. Let R be the set of non-omitted jobs in S'. W.l.o.g., we can assume that S' executes each job in R at a uniform speed, as this minimizes the energy consumption.

For every $j \in R$, define $\mathrm{SMALL}(j) := \{x \in J \setminus R : \mathrm{big}(x) = j\}$. Note that every omitted job occurs in exactly one of the sets $\mathrm{SMALL}(j)$, $j \in R$. The schedule S is constructed as follows. For all $j \in R$, we process the jobs $\{j\} \cup \mathrm{SMALL}(j)$ using the execution interval of j in S' and a uniform speed. The processing order may be chosen arbitrarily. Clearly, the resulting schedule is feasible by the definition of $\mathrm{big}(x)$. In order to finish the total volume V_j of the jobs $\{j\} \cup \mathrm{SMALL}(j)$ within the interval of j in S', we need to raise the speed in this interval by the factor V_j/v_j. As $|\mathrm{SMALL}(j)| \leq n$, and $v_x \leq v_j(1+\epsilon)^{-\delta}$ for all $x \in \mathrm{SMALL}(j)$, we have that

$$V_j \leq v_j + nv_j(1+\epsilon)^{-\delta} \leq v_j + nv_j\frac{\epsilon}{n} \leq (1+\epsilon)v_j,$$

where the second inequality follows from the definition of δ. For the speedup factor, we therefore obtain $V_j/v_j \leq 1+\epsilon$. Hence, the energy consumption grows by at most the factor $(1+\epsilon)^\alpha$. □

The preceding lemma essentially ensures that representing the δ largest size classes of an "inherited" job set suffices if we allow a small increment of energy cost. The smaller jobs can then be added safely (i.e. without increasing the energy cost by too much) to the final schedule. We now turn to the central definition of the dynamic program. All schedules in this definition are with respect to the rounded instance $\mathcal{I}'$, and all grid points relate to the set $\mathcal{P}_{\text{approx}}$.

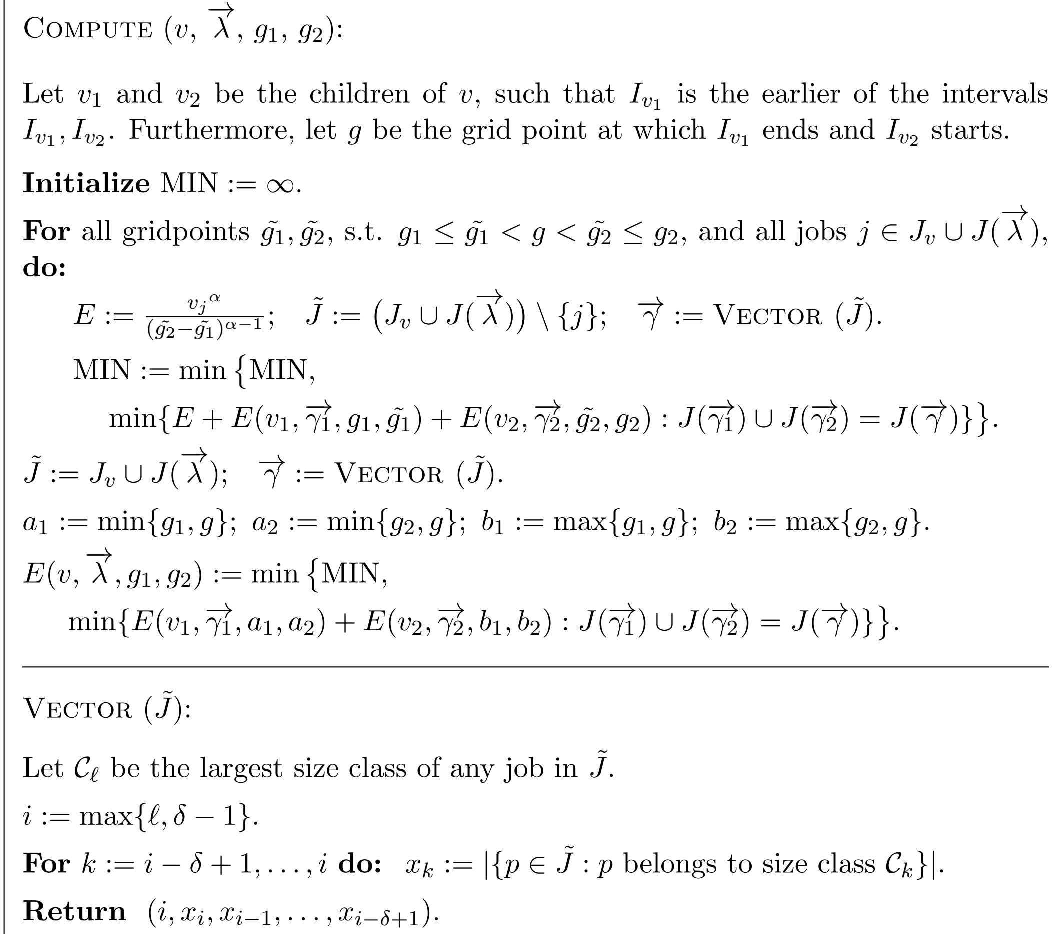
Definition 2.3.8. *For any vertex v in the tree T, any job vector $\overrightarrow{\lambda}$ that is heritable to v, and any pair of grid points $g_1 \leq g_2$ with $[g_1, g_2) \subseteq I_v$, let $G(v, \overrightarrow{\lambda}, g_1, g_2)$ denote a minimum cost grid point schedule for the jobs in the subtree of v (including v itself) plus the jobs $J(\overrightarrow{\lambda})$ (these are allowed to be scheduled anywhere inside $[g_1, g_2)$) that uses only the interval $[g_1, g_2)$. Furthermore, let $S(v, \overrightarrow{\lambda}, g_1, g_2)$ be a δ-omitted schedule for the same set of jobs in the same interval $[g_1, g_2)$, satisfying $E\big(S(v, \overrightarrow{\lambda}, g_1, g_2)\big) \leq E\big(G(v, \overrightarrow{\lambda}, g_1, g_2)\big)$.*

Dynamic Program. Our dynamic program computes the schedules $S(v, \overrightarrow{\lambda}, g_1, g_2)$. For ease of exposition, we focus only on computing the energy consumption values $E(v, \overrightarrow{\lambda}, g_1, g_2) := E\big(S(v, \overrightarrow{\lambda}, g_1, g_2)\big)$, and omit the straightforward bookkeeping of the corresponding schedules. The base cases are the leaves of T. For a particular leaf node ℓ, we set

$$E(\ell, \overrightarrow{\lambda}, g_1, g_2) := \begin{cases} 0 & \text{if } J_\ell \cup J(\overrightarrow{\lambda}) = \emptyset \\ \frac{V^\alpha}{(g_2-g_1)^{\alpha-1}} & \text{otherwise,} \end{cases}$$

where V is the total volume of all jobs in $J_\ell \cup J(\overrightarrow{\lambda})$. This corresponds to executing $J_\ell \cup J(\overrightarrow{\lambda})$ at uniform speed using the whole interval $[g_1, g_2)$. The resulting schedule is feasible, as no release times or deadlines occur in the interior of I_ℓ. Furthermore, it is also optimal by the convexity of the power function. Thus $E(\ell, \overrightarrow{\lambda}, g_1, g_2) \leq E\big(G(\ell, \overrightarrow{\lambda}, g_1, g_2)\big)$.

When all leaves have been handled, we move on to the next level, the parents of the leaves. For this and also the following levels up to the root r, we compute the values $E(v, \overrightarrow{\lambda}, g_1, g_2)$ recursively, using the procedure COMPUTE in Figure 2.2. An intuitive description of the procedure is given below.



COMPUTE $(v, \overrightarrow{\lambda}, g_1, g_2)$:

Let v_1 and v_2 be the children of v, such that I_{v_1} is the earlier of the intervals I_{v_1}, I_{v_2}. Furthermore, let g be the grid point at which I_{v_1} ends and I_{v_2} starts.

Initialize MIN $:= \infty$.

For all gridpoints $\tilde{g_1}, \tilde{g_2}$, s.t. $g_1 \le \tilde{g_1} < g < \tilde{g_2} \le g_2$, and all jobs $j \in J_v \cup J(\overrightarrow{\lambda})$, **do:**

$E := \frac{{v_j}^\alpha}{(\tilde{g_2}-\tilde{g_1})^{\alpha-1}}$; $\tilde{J} := \big(J_v \cup J(\overrightarrow{\lambda})\big) \setminus \{j\}$; $\overrightarrow{\gamma} :=$ VECTOR $(\tilde{J})$.

MIN $:= \min\big\{$MIN,

$\min\{E + E(v_1, \overrightarrow{\gamma_1}, g_1, \tilde{g_1}) + E(v_2, \overrightarrow{\gamma_2}, \tilde{g_2}, g_2) : J(\overrightarrow{\gamma_1}) \cup J(\overrightarrow{\gamma_2}) = J(\overrightarrow{\gamma})\}\big\}$.

$\tilde{J} := J_v \cup J(\overrightarrow{\lambda})$; $\overrightarrow{\gamma} :=$ VECTOR $(\tilde{J})$.

$a_1 := \min\{g_1, g\}$; $a_2 := \min\{g_2, g\}$; $b_1 := \max\{g_1, g\}$; $b_2 := \max\{g_2, g\}$.

$E(v, \overrightarrow{\lambda}, g_1, g_2) := \min\big\{$MIN,

$\min\{E(v_1, \overrightarrow{\gamma_1}, a_1, a_2) + E(v_2, \overrightarrow{\gamma_2}, b_1, b_2) : J(\overrightarrow{\gamma_1}) \cup J(\overrightarrow{\gamma_2}) = J(\overrightarrow{\gamma})\}\big\}$.

VECTOR $(\tilde{J})$:

Let $\mathcal{C}_\ell$ be the largest size class of any job in $\tilde{J}$.

$i := \max\{\ell, \delta - 1\}$.

For $k := i - \delta + 1, \ldots, i$ **do:** $x_k := |\{p \in \tilde{J} : p$ belongs to size class $\mathcal{C}_k\}|$.

Return $(i, x_i, x_{i-1}, \ldots, x_{i-\delta+1})$.

Figure 2.2.: Procedure for computing the remaining entries of the DP.

Our first step is to iterate through all possible options for a potential "crossing" job j, whose execution interval $[\tilde{g_1}, \tilde{g_2})$ stretches from child v_1 into the interval of child v_2. For every possible choice, we combine the optimal energy cost E for this job (obtained by using a uniform execution speed) with the best possible way to split up the remaining jobs between the truncated intervals of v_1 and v_2. Here we consider only the δ largest size classes of the remaining jobs $\tilde{J}$, and omit the smaller jobs. This omitting happens during the construction of a vector representation for $\tilde{J}$ using the procedure VECTOR. Finally, we also try the option that no "crossing" job exists and all jobs are split up between v_1 and v_2. In this case we need to take special care of the subproblem boundaries, as $g_1 > g$ or $g_2 < g$ are also valid arguments for the

procedure Compute.

Lemma 2.3.9. *The schedules $S(v, \overrightarrow{\lambda}, g_1, g_2)$ constructed by the above dynamic program are δ-omitted schedules for the jobs in the subtree of v plus the jobs $J(\overrightarrow{\lambda})$. Furthermore, they satisfy $E\big(S(v, \overrightarrow{\lambda}, g_1, g_2)\big) \leq E\big(G(v, \overrightarrow{\lambda}, g_1, g_2)\big)$.*

Proof. We prove the lemma by induction. In the base cases, that is at the leaves of T, we already argued that the schedules are feasible and optimal. Since no jobs are omitted at all, the lemma is obviously true at this level. We now perform the induction step. To this end, let us consider a fixed schedule $S(v, \overrightarrow{\lambda}, g_1, g_2)$, and let us assume the lemma is true for the children v_1 and v_2 of v. We first show that $S(v, \overrightarrow{\lambda}, g_1, g_2)$ is indeed a δ-omitted schedule. The only point where jobs are omitted in the recursive procedure is the call of Vector $(\tilde{J})$, where a vector-representation $\overrightarrow{\gamma}$ of $\tilde{J}$ is constructed. This vector $\overrightarrow{\gamma}$ only represents a subset of the jobs $\tilde{J}$, namely the jobs in the δ largest size classes of $\tilde{J}$. Let $j_{\max}$ denote a job in $\tilde{J}$ with maximum volume, i.e. a job belonging to the largest size class. Then every omitted job j_{om} has volume at most $v_{j_{\max}}(1+\epsilon)^{-\delta}$, and we can choose $\text{big}(j_{om}) := j_{\max}$ to satisfy the requirements of Definition 2.3.6, as long as $j_{\max}$ is indeed contained in one of the subschedules that we combine in the recursion step. If, however, $j_{\max}$ is omitted in the corresponding subschedule, then there exists a job $\text{big}(j_{\max})$ as required in Definition 2.3.6, by induction hypothesis. In this case we can choose $\text{big}(j_{om}) := \text{big}(j_{\max})$. This proves that $S(v, \overrightarrow{\lambda}, g_1, g_2)$ is indeed a δ-omitted schedule.

We now argue about the energy consumption. Let J_1 and J_2 denote the subsets of jobs that $G(v, \overrightarrow{\lambda}, g_1, g_2)$ processes entirely within I_{v_1} and I_{v_2}, respectively. If there exists a "crossing" job spanning from I_{v_1} into I_{v_2}, we denote this job by j_c. Now we look at the iteration that handles exactly this situation, i.e. when $j = j_c$ and $\tilde{g_1}, \tilde{g_2}$ mark the beginning and end of j_c's execution interval in $G(v, \overrightarrow{\lambda}, g_1, g_2)$, or the passage after the for-loop for the case without "crossing" job. As mentioned earlier, the procedure possibly omits certain jobs and only splits up a subset of $J_v \cup J(\overrightarrow{\lambda})$ between the children v_1 and v_2. Since all possible splits are tried, one option for the min-operation is to combine subschedules that process a subset of J_1 within I_{v_1}, and a subset of J_2 within I_{v_2}. By induction hypothesis, and since we only schedule subsets of J_1 and J_2, the energy consumption of these subschedules is at most the energy spent by $G(v, \overrightarrow{\lambda}, g_1, g_2)$ for executing J_1 and J_2, respectively. Furthermore, if there exists a "crossing" job j_c, then executing this job from $\tilde{g_1}$ to $\tilde{g_2}$ at uniform speed costs at most the energy that $G(v, \overrightarrow{\lambda}, g_1, g_2)$ pays for this job. Summing up the different parts, we get that the considered option has an energy consumption of at most $E\big(G(v, \overrightarrow{\lambda}, g_1, g_2)\big)$. The lemma follows as we choose the minimum over all possible options. □

Combining Lemmas 2.2.4, 2.3.2, 2.3.7, and 2.3.9 we can now state our main theorem.

Theorem 2.3.10. *The non-preemptive speed scaling problem admits a QPTAS if the instance is laminar.*

Proof. Let r^* be the earliest release time, and d^* be the latest deadline of any job in $\mathcal{J}$. Furthermore, let r be the root of the tree T, and let $\overrightarrow{0}$ denote the (heritable) job vector representing the empty set, i.e. $\overrightarrow{0} := (\delta - 1, 0, \ldots, 0)$. We consider the schedule $S(r, \overrightarrow{0}, r^*, d^*)$, which is a δ-omitted schedule for the rounded instance by Lemma 2.3.9, and turn it into a feasible schedule S_r for the whole set of (rounded) jobs, using Lemma 2.3.7. Finally, we apply Lemma 2.3.2 to turn S_r into a feasible schedule S for the original instance $\mathcal{I}$, and obtain

$$\begin{aligned} E(S) &\leq (1+\epsilon)^{\alpha} E(S_r) \leq (1+\epsilon)^{2\alpha} E\big(S(r, \overrightarrow{0}, r^*, d^*)\big) \\ &\leq (1+\epsilon)^{2\alpha} E\big(G(r, \overrightarrow{0}, r^*, d^*)\big) \leq (1+\epsilon)^{3\alpha-1}\mathrm{OPT} = \big(1+\mathcal{O}(\epsilon)\big)\mathrm{OPT}. \end{aligned}$$

Here, the third inequality holds by Lemma 2.3.9, and the fourth inequality follows from Lemma 2.2.4 and the fact that $G(r, \overrightarrow{0}, r^*, d^*)$ is an optimal grid point schedule for the rounded instance (with smaller job volumes). The quasipolynomial running time of the algorithm is easily verified, as we have only a polynomial number of grid points, and at most a quasipolynomial number of job vectors that are heritable to any vertex of the tree. □

2.4. Equal-Volume Jobs

In this section, we consider the case that all jobs have the same volume $v_1 = v_2 = \ldots = v_n = v$. We present a dynamic program that computes an (exact) optimal schedule for this setting in polynomial time. All grid points used for this purpose relate to the set $\mathcal{P}_{\text{exact}}$.

As a first step, let us order the jobs such that $r_1 \leq r_2 \leq \ldots \leq r_n$. Furthermore, let us define an ordering on schedules as follows.

Definition 2.4.1 (Completion Time Vector)**.** *Let $C_1, \ldots, C_n$ be the completion times of the jobs $j_1, \ldots, j_n$ in a given schedule S. The vector $\overrightarrow{S} := (C_1, \ldots, C_n)$ is called the* completion time vector *of S.*

Definition 2.4.2 (Lexicographic Ordering)**.** *A schedule S is said to be* lexicographically smaller *than a schedule S' if the first component in which their completion time vectors differ is smaller in $\overrightarrow{S}$ than in $\overrightarrow{S'}$.*

We now elaborate the details of the DP, focusing on energy consumption values only.

Definition 2.4.3. *Let $i \in \{1, \ldots, n\}$ be a job index, and let $g_1, g_2,$ and g_3 be grid points satisfying $g_1 \leq g_2 \leq g_3$. We define $E(i, g_1, g_2, g_3)$ to be the minimum energy consumption of a grid point schedule for the jobs $\{j_k \in \mathcal{J} : k \geq i \ \wedge \ g_1 < d_k \leq g_3\}$ that uses only the interval $[g_1, g_2)$.*

Dynamic Program. Our goal is to compute the values $E(i, g_1, g_2, g_3)$. To this end, we let

$$E(i, g_1, g_2, g_3) := \begin{cases} 0 & \text{if } \{j_k \in \mathcal{J} : k \geq i \ \wedge \ g_1 < d_k \leq g_3\} = \emptyset \\ \infty & \text{if } \ \exists k \geq i : g_1 < d_k \leq g_3 \ \wedge \ [r_k, d_k) \cap [g_1, g_2) = \emptyset. \end{cases}$$

Note that if $g_1 = g_2$, one of the above cases must apply. We now recursively compute the remaining values, starting with the case that g_1 and g_2 are consecutive grid points, and stepwise moving towards cases with more and more grid points in between g_1 and g_2. The recursion works as follows. Let $E(i, g_1, g_2, g_3)$ be the value we want to compute, and let j_q be the smallest index job in $\{j_k \in \mathcal{J} : k \geq i \ \wedge \ g_1 < d_k \leq g_3\}$. Furthermore, let $\mathcal{G}$ denote a lexicographically smallest optimal grid point schedule for the jobs $\{j_k \in \mathcal{J} : k \geq i \wedge g_1 < d_k \leq g_3\}$, using only the interval $[g_1, g_2)$. Our first step is to "guess" the grid points b_q and e_q that mark the beginning and end of j_q's execution interval in $\mathcal{G}$, by minimizing over all possible options. We then use the crucial observation that in $\mathcal{G}$, all jobs $J^- := \{j_k \in \mathcal{J} : k \geq q+1 \ \wedge \ g_1 < d_k \leq e_q\}$ are processed completely before j_q, and all jobs $J^+ := \{j_k \in \mathcal{J} : k \geq q+1 \ \wedge \ e_q < d_k \leq g_3\}$ are processed completely after j_q. For J^- this is obviously the case because of the deadline constraint. For J^+ this holds as all these jobs have release time at least r_q by the ordering of the jobs, and deadline greater than e_q by definition of J^+. Therefore any job in J^+ that is processed before j_q could be swapped with j_q, resulting in a lexicographic smaller schedule; a contradiction. Hence, we can use the following recursion to compute $E(i, g_1, g_2, g_3)$.

$$E(i, g_1, g_2, g_3) := \min \Big\{ \frac{{v_q}^{\alpha}}{(e_q - b_q)^{\alpha - 1}} + E(q+1, g_1, b_q, e_q) + E(q+1, e_q, g_2, g_3) : \\ (g_1 \leq b_q < e_q \leq g_2) \wedge (b_q \geq r_q) \wedge (e_q \leq d_q) \Big\}.$$

Once we have computed all values, we output the schedule S corresponding to $E(1, r^*, d^*, d^*)$, where r^* is the earliest release time and d^* the latest deadline of any job in $\mathcal{J}$. Lemma 2.2.6 implies that $E(S) = \text{OPT}$.

Theorem 2.4.4. *The non-preemptive speed scaling problem admits a polynomial time algorithm if all jobs have the same volume.*

2.5. Purely-Laminar Instances

In this section, we present an FPTAS for a purely-laminar instance $\mathcal{I}$. W.l.o.g., we assume that the jobs are ordered by inclusion of their time windows, that is $[r_1, d_1) \subseteq [r_2, d_2) \subseteq \cdots \subseteq [r_n, d_n)$. Furthermore, whenever we refer to grid points in this section, we refer to the set $\mathcal{P}_{\text{approx}}$. Our FPTAS uses dynamic programming to construct an optimal grid point schedule for $\mathcal{I}$, satisfying the following structural property:

Property 2.5.1. *For any $k > 1$, jobs $j_1, \dots, j_{k-1}$ are either all processed before j_k, or all processed after j_k.*

This structure can easily be established in any schedule for $\mathcal{I}$ by performing a sequence of energy-preserving swaps. According to this, the following lemma is a straightforward extension of Lemma 2.2.4 to the purely-laminar case.

Lemma 2.5.2. *If the problem instance is purely-laminar, there exists a grid point schedule $\mathcal{G}$ with respect to $\mathcal{P}_{\text{approx}}$ that satisfies Property 2.5.1 and has energy cost $E(\mathcal{G}) \leq (1+\epsilon)^{\alpha-1}\text{OPT}$.*

Proof. Consider an optimal schedule $\mathcal{S}^*$ for $\mathcal{I}$, and let J^- and J^+ be the jobs executed before and after j_1, respectively. Now rearrange the execution intervals (without changing their lengths) of the jobs in J^+ into *smallest index first* order (SIF), by repeatedly swapping two consecutively processed jobs j_a preceding j_b, with $a > b$. For the swap, we let the execution interval of j_b now start at j_a's original starting time, and directly append j_a's execution interval once j_b is finished. Note that each such swap maintains feasibility, as no release times occurs during the execution of the jobs in J^+, and $a > b$ implies $d_a \geq d_b$. Similarly, we rearrange the execution intervals of the jobs in J^- into *largest index first* order (LIF), and denote the resulting schedule by S'. Clearly, $E(S') = \text{OPT}$, as the rearrangements preserve the energy cost of every individual job. Furthermore, S' satisfies Property 2.5.1. To see this, let us fix $k > 1$ and distinguish whether j_k is in J^- or in J^+. In the first case, when $j_k \in J^-$, all $j \in J^+$ are scheduled after j_k by definition of J^-/J^+, and all $j_i \in J^-, i < k$ are scheduled after j_k by the LIF-order. In the second case, when $j_k \in J^+$, all $j \in J^-$ are scheduled before j_k by definition of J^-/J^+, and all $j_i \in J^+, i < k$ are scheduled before j_k by the SIF-order. As a final step, we now apply the transformation from the proof of Lemma 2.2.4 to S'. Since this transformation does not change the order of any jobs, the resulting grid point schedule $\mathcal{G}$ still satisfies Property 2.5.1, and has energy cost $E(\mathcal{G}) \leq (1+\epsilon)^{\alpha-1}\text{OPT}$. □

Dynamic Program. For any $k \leq n$ and grid points $g_1 \leq g_2$, let $S(k, g_1, g_2)$ denote a minimum cost grid point schedule for $j_1, \dots, j_k$ that satisfies Property 2.5.1 and uses only the time interval between g_1 and g_2. The corresponding energy cost of $S(k, g_1, g_2)$ is denoted by $E(k, g_1, g_2)$, where $E(k, g_1, g_2) := \infty$ if no such schedule exists. For ease of exposition, we only show how to compute the energy consumption values $E(k, g_1, g_2)$, and omit the straightforward bookkeeping of the corresponding schedules. The base cases are given by $E(0, g_1, g_2) = 0$, for all $g_1 \leq g_2$. All remaining entries can be computed with the following recursion.

$$E(k+1, g_1, g_2) = \begin{cases} \infty \quad \text{if } (g_1 = g_2) \vee (g_1 \geq d_{k+1}) \vee (g_2 \leq r_{k+1}). \\ \min\left\{ \frac{{v_{k+1}}^\alpha}{(g_2'-g_1')^{\alpha-1}} + \min\{E(k, g_1, g_1'), E(k, g_2', g_2)\} : \right. \\ \qquad \left. (g_1 \leq g_1' < g_2' \leq g_2) \wedge (g_1' \geq r_{k+1}) \wedge (g_2' \leq d_{k+1}) \right\} \\ \qquad\qquad \text{otherwise.} \end{cases}$$

Intuitively, we minimize over all possible combinations of grid points g'_1 and g'_2 that could mark the beginning and end of j_{k+1}'s execution. For fixed g'_1 and g'_2, it is best to process j_{k+1} at uniform speed, resulting in the energy cost ${v_{k+1}}^\alpha/(g'_2 - g'_1)^{\alpha-1}$ for this job. The remaining jobs $j_1, \ldots, j_k$ must then be scheduled either before or after j_{k+1}, to satisfy Property 2.5.1. This fact is captured in the second min-operation of the formula. The constraints on g'_1 and g'_2 ensure that j_{k+1} can be feasibly scheduled in the chosen interval.

Once we have computed all values $E(k, g_1, g_2)$ (and their corresponding schedules), we output the schedule $\tilde{S} := S(n, r^*, d^*)$, where r^* is the earliest release time and d^* the latest deadline of any job in $\mathcal{I}$. Note that $\tilde{S}$ is an optimal grid point schedule with Property 2.5.1 for $\mathcal{I}$. Hence, Lemma 2.5.2 implies that $E(\tilde{S}) \leq (1+\epsilon)^{\alpha-1}\mathrm{OPT} = \big(1+\mathcal{O}(\epsilon)\big)\mathrm{OPT}$. Finally, it is easy to verify that the running time of the algorithm is polynomial in n and $1/\epsilon$, since the total number of grid points in $\mathcal{P}_{\text{approx}}$ is $\mathcal{O}\big(n^3(1+\frac{1}{\epsilon})\big)$. We therefore obtain the following theorem.

Theorem 2.5.3. *The non-preemptive speed scaling problem admits an FPTAS if the instance is purely-laminar.*

2.6. Bounded Number of Time Windows

Let us consider a problem instance $\mathcal{I}$, and group together jobs that share the same time window. We refer to the group of jobs with time window $[r, d)$ as the *type* T_{rd}.

Theorem 2.6.1. *The non-preemptive speed scaling problem admits an FPTAS if the total number of types is at most a constant c.*

Our FPTAS draws on ideas of Antoniadis and Huang [8], as we transform the problem into an instance $\mathcal{I}'$ of *unrelated machine scheduling* with ℓ_α-norm objective. In this problem, one is given a set of machines $\mathcal{M}$, a set of jobs $\mathcal{J}$, and numbers p_{ij} that specify the processing time of job j on machine i. The goal is to find an assignment A of the jobs to the machines that minimizes $\mathrm{Cost}(A) = (\sum_{i\in\mathcal{M}}(\sum_{j:A(j)=i} p_{ij})^\alpha)^{1/\alpha}$. In general, this problem is APX-hard [12]. Our instance, however, will have only a constant number of machines, and for this special case an FPTAS exists [12].

The transformation works as follows. Let $\mathcal{G}$ be an optimal grid point schedule with respect to $\mathcal{P}_{\text{approx}}$, and for each type T_{rd} let $b(T_{rd})$ and $e(T_{rd})$ denote the grid points at which $\mathcal{G}$ starts to process the first job of T_{rd} and finishes the last job of T_{rd}, respectively. Our first step is to "guess" the entire set of grid points $b(\cdot)$ and $e(\cdot)$, by trying out all possible options with $r \leq b(T_{rd}) < e(T_{rd}) \leq d$ for every type T_{rd}. Note that the total number of choices that we have to make is at most $\mathcal{O}\big(n^{6c}(1+\frac{1}{\epsilon})^{2c}\big)$, and thus polynomial in both n and $1/\epsilon$. For one particular guess, let $g_1 < g_2 < \ldots < g_k$ be the ordered set of distinct grid points $b(\cdot)$ and $e(\cdot)$. The instance $\mathcal{I}'$ has a machine i for every interval $[g_i, g_{i+1})$, and a job j for every job of the original instance. The processing times p_{ij} are given as $p_{ij} := \frac{v_j}{(g_{i+1}-g_i)^{1-1/\alpha}}$ if $[g_i, g_{i+1}) \subseteq [r_j, d_j)$, and $p_{ij} := \infty$ otherwise.

Note that the total number of machines in $\mathcal{I}'$ is $k-1 < 2c$. Hence, the FPTAS of [12] can be applied to obtain an assignment A with $\text{Cost}(A) \le (1+\epsilon)\text{OPT}'$, where OPT' denotes the cost of an optimal assignment for $\mathcal{I}'$. The following two lemmas imply Theorem 2.6.1.

Lemma 2.6.2. *Every finite-cost assignment A for $\mathcal{I}'$ can be transformed into a schedule S for $\mathcal{I}$, such that $E(S) = \big(\text{Cost}(A)\big)^{\alpha}$.*

Proof. For any $i \in \mathcal{M}$, let A_i denote the set of jobs that A assigns to machine i. In order to create the schedule S, we iterate through all $i \in \mathcal{M}$ and process the jobs in A_i within the interval $[g_i, g_{i+1})$, using the uniform speed $(\sum_{j\in A_i} v_j)/(g_{i+1}-g_i)$. The resulting schedule is clearly feasible, as A has finite cost and every $j \in A_i$ thus satisfies $[g_i, g_{i+1}) \subseteq [r_j, d_j)$. For the energy consumption of S we get

$$E(S) = \sum_{i\in\mathcal{M}} \left(\frac{\sum_{j\in A_i} v_j}{g_{i+1}-g_i}\right)^{\alpha} (g_{i+1}-g_i) = \sum_{i\in\mathcal{M}} \left(\frac{\sum_{j\in A_i} v_j}{(g_{i+1}-g_i)^{1-1/\alpha}}\right)^{\alpha}$$

$$= \sum_{i\in\mathcal{M}} \left(\sum_{j\in A_i} p_{ij}\right)^{\alpha} = \Big(\text{Cost}(A)\Big)^{\alpha}.$$

□

Lemma 2.6.3. *If the grid points $b(\cdot)$ and $e(\cdot)$ are guessed correctly, there exists an assignment A for $\mathcal{I}'$ with $\text{Cost}(A) \le \big((1+\epsilon)^{\alpha-1}\text{OPT}\big)^{1/\alpha}$.*

Proof. Remember that $\mathcal{G}$ is an optimal grid point schedule for $\mathcal{I}$, and that the grid points $b(T_{rd})$ and $e(T_{rd})$ mark the time points at which $\mathcal{G}$ starts to process the first job of type T_{rd} and finishes the last job of T_{rd}, respectively. Now observe that in $\mathcal{G}$, every job j is processed entirely within some interval $[g_i, g_{i+1})$, satisfying $[g_i, g_{i+1}) \subseteq [r_j, d_j)$. This is true because $r_j \le b(T_{r_j d_j}) < e(T_{r_j d_j}) \le d_j$, and no job can stretch from an interval $[g_{x-1}, g_x)$ into $[g_x, g_{x+1})$ since g_x indeed marks the beginning or end of some job. Let A_i denote the set of jobs which are entirely processed within $[g_i, g_{i+1})$, and let A be the assignment that maps all jobs from A_i to machine i. The cost of A is given as

$$\text{Cost}(A) = \left(\sum_{i\in\mathcal{M}}\Big(\sum_{j\in A_i} p_{ij}\Big)^{\alpha}\right)^{1/\alpha} = \left(\sum_{i\in\mathcal{M}}\Big(\frac{\sum_{j\in A_i} v_j}{(g_{i+1}-g_i)^{1-1/\alpha}}\Big)^{\alpha}\right)^{1/\alpha}$$

$$= \left(\sum_{i\in\mathcal{M}}\Big(\frac{\sum_{j\in A_i} v_j}{g_{i+1}-g_i}\Big)^{\alpha}(g_{i+1}-g_i)\right)^{1/\alpha}$$

$$\le \Big(E(\mathcal{G})\Big)^{1/\alpha} \le \Big((1+\epsilon)^{\alpha-1}\text{OPT}\Big)^{1/\alpha}.$$

Here the last two inequalities follow from the convexity of the power function and Lemma 2.2.4, respectively. □

3. Speed Scaling with Variable Electricity Rates and Speed Limits

3.1. Introduction

In this chapter, we study an extension of Yao et al.'s original speed scaling model [79]. The focus of our work lies in two aspects:

Dynamic Speed Limits: In the literature, most work adopts the unbounded speed model of [79], where the processing speed can be arbitrarily large. In practice, however, there are limits to the speed, and they no longer stem solely from the maximum processor frequency. Instead, as devices become smaller and more sensitive to environmental conditions like temperature and humidity, speed limits become highly dynamic. For example, failures of air conditioning, broken fans, or airflow problems can cause severe temperature fluctuations in data centers [51], making it necessary to temporarily slow down one or more processors. Another cause for highly dynamic speed limits are voltage fluctuations: e.g., the power input of solar-powered devices depends heavily on the current sun exposure, and even the supply through normal power outlets can vary with the time of the day.

Dynamic Electricity Costs: A second, often neglected model constraint, are dynamic electricity costs. In particular for data centers, energy minimization aims at cost reduction. But often, algorithm design assumes energy costs to be uniform over time. However, electricity providers increasingly adopt time-dependent tariff policies. In fact, most providers already offer heavily discounted rates during off peak times, for example at night or before noon. While such cost changes are not as frequent and dynamic as the aforementioned changes of the maximum speed, they can have a huge impact on the operating costs.

Our Contribution. We extend the standard model of Yao et al. [79] by the above discussed aspects of dynamic speed limits and electricity costs. That is, we consider the scheduling of jobs on a single, speed-scalable processor. Each job comes with its own release time, deadline, and workload (volume). We assume a limit on the processor's maximum speed, which varies over time (to cover the above mentioned dynamics, we allow almost arbitrary, even continuous constraints). Furthermore, we also allow energy prices to vary over time. Our goal is to find a schedule that minimizes the total energy costs. For this problem, we provide a polynomial-time optimal algorithm. Even though we use convex programming in our analysis, the derived algorithm is purely combinatorial.

An important aspect of our problem is that standard techniques are rendered infeasible by the infinite number of variables/constraints (because of the continuous constraint functions). The typical approach to deal with such issues in algorithm design is to discretize the problem and use the well-known KKT conditions (e.g., [9]). A major contribution of our work is to exemplify how calculus of variations can be used to deal with continuous constraints in a combinatorial way. For this, we derive an optimality condition similar to the well-known KKT conditions, but extended to continuous constraints. To the best of our knowledge, there is no previous work on such an explicit extension of the KKT conditions to the continuous case.

Related Work. Special cases of both the maximum speed and the electricity tariff setting have been studied before. Chan et al. [32] and Li [60] assume that there is a constant upper bound on the available speed, and one wants to maximize the throughput of the schedule while minimizing the energy consumption. On the other hand, Fang et al. [38] consider electricity tariffs, but without an upper bound on the speed and in a much more restricted setting: their model is equivalent to considering only one-job-instances and discrete dynamics in our problem. They develop an optimal polynomial-time algorithm, by a technique which to some extent resembles ours. However, since we consider a significantly more general setting, we have to cope with several important aspects not appearing in [38], in particular we need to extend the KKT conditions using variational calculus. Electricity tariffs have also been considered beyond the speed-scaling setting, see for example [56]. Further, Thang [74] uses the lagrangian dual of a mathematical program in order to analyze several online scheduling algorithms with flow-time objectives. Although [74] also has the same view of optimizing over a set of arbitrary speed functions, it differs from our approach in that lagrangian duality is used more as a tool for analyzing the approximation ratio, rather than for characterizing an optimal solution and deriving an optimal algorithm. Finally, Bansal et al. [22] consider a speed scaling problem where energy is supplied at a limited rate. However, their supply rate is constant over time and they seek to minimize this rate, rather than the overall energy consumption. See [1] and [48] for broader surveys on energy-efficient algorithms.

3.2. Model and Preliminaries

We consider the scheduling of n jobs $J \coloneqq \{ 1, 2, \ldots, n \}$ on a single, speed-scalable processor. Here, speed-scalable means that the processor's speed $s \in \mathbb{R}_{\geq 0}$ is controlled by the scheduler. The power consumption is modeled by a *power function* $P \colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}, s \mapsto s^{\alpha}$. That is, while running at speed s, energy is consumed at a rate of $P(s) = s^{\alpha}$. The constant $\alpha > 1$ is called the *energy exponent*. In addition to these classical speed scaling properties, we have the constraint that the maximal speed at time t is bounded. We model this constraint via a *maximum speed function* $s_{\max} \colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$. Further, there is a cost factor associated with every time point $t \in \mathbb{R}_{\geq 0}$, specifying the cost per unit of energy. The cost factor is modeled via a *cost*

factor function $c\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{>0}$.

Each job $j \in J$ comes with a *release time* $r_j \in \mathbb{R}_{\geq 0}$, a *deadline* $d_j \in \mathbb{R}_{\geq 0}$, and a *workload* $w_j \in \mathbb{R}_{\geq 0}$. For each time $t \in \mathbb{R}_{\geq 0}$, a *schedule* S must decide which job to process at what speed. Preemption is allowed, so that a job may be suspended and resumed later on. We model a schedule S by a *speed function* $s\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ and a *scheduling policy* $\mathcal{J}\colon \mathbb{R}_{\geq 0} \to J$. Here, $s(t)$ denotes the speed at time t, and $\mathcal{J}(t)$ the job that is scheduled at time t. A *feasible* schedule must finish all jobs within their release time/deadline intervals $[r_j, d_j)$ without exceeding the maximum speed function $s_{\max}$. More formally, we require $s(t) \leq s_{\max}(t)$ for all $t \in \mathbb{R}_{\geq 0}$ and $\int_{\mathcal{J}^{-1}(j)\cap[r_j,d_j)} s(t)\,\mathrm{d}t \geq w_j$ for all $j \in J$. The total energy consumption of a schedule S is given by $\int_0^\infty P(s(t))\,\mathrm{d}t$, and its total energy cost by $E(s) \coloneqq \int_0^\infty c(t)\cdot P(s(t))\,\mathrm{d}t$. For technical reasons, we restrict ourselves to functions in C_{pr} (i.e., to functions that are right-continuous with finitely many discontinuities), which covers all practically relevant schedules. Our goal is to find a feasible schedule of minimum cost. We refer to this scheduling problem as ContBERS (**Cont**inuous **B**ounded Speed & **E**lectricity **R**ates **S**cheduling).

Computational Model. We assume oracle access to the functions $s_{\max}$ and c. Similarly, we assume access to basic function calculus like taking the min of two functions or computing integrals (cf. Section 3.4.1). This is in accord with standard speed-scaling literature (e.g., [4, 25]) where one needs the ability to, for example, solve equations involving high degree polynomials.

3.3. Balance for Optimality

This section is dedicated to proving the following theorem.

Theorem 3.3.1. *A feasible schedule is optimal if and only if it is both non-wasting and work-balanced.*

The properties *non-wasting* and *work-balanced* are natural structural properties, which we formally introduce in Section 3.3.3. For now, think of schedules that distribute the jobs' workload "as evenly as possible" while taking constraints (e.g., the release times/deadlines or the speed limits) and cost factors into account.

Being work-balanced is a natural condition, and similar structural properties have been exploited for a variety of problems to study and compute optimal or approximate solutions. Examples include the original speed-scaling algorithm YDS [18], the standard approximation algorithm for metric facility location [52], or the study of equilibria in resource selection games [44]. These properties are typically derived using a linear or convex program and duality theory, and eventually yield a corresponding *primal-dual algorithm.* The basic ingredients for such an approach are the KKT conditions known from convex programming [29]. Unfortunately, this approach does not work in our setting. Although the considered optimization problem

is convex, the general maximum speed restriction leads to infinitely many variables/-constraints: for each time $t \in \mathbb{R}_{\geq 0}$ the speed must not exceed $s_{\max}(t)$. Based on the theory of variational calculus [72], we can still approach the ContBERS problem by similar means. While variational calculus is a pretty common toolkit (e.g., in the area of operations research), it is mostly used for solving optimization problems directly via numerical methods. Instead, we use it as an analysis tool to derive "extended KKT conditions", yielding a structural characterization of optimal solutions. As typical for primal-dual approaches, this leads to a quite intuitive algorithm.

Overview. We continue in Section 3.3.1 with a presentation of the ContBERS problem viewed as an optimization problem with an infinite number of constraints. Afterward, Section 3.3.2 provides a framework to characterize optimal solutions for problems of a more general form. Finally, Section 3.3.3 applies this framework to prove Theorem 3.3.1.

3.3.1. Scheduling via Variational Calculus

Optimization problems with infinitely many variables/constraints can be modeled via function variables. In the case of the ContBERS problem, one can think of the schedule's speed function $s\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ as a variable that has to fulfill the constraint $s(t) \leq s_{\max}(t)$ at any time $t \geq 0$. Remember that the costs of a speed function s are given by $E(s) = \int_0^\infty c(t)P(s(t))\,\mathrm{d}t \in \mathbb{R}$. In other words, E is a function that maps a speed function s to a real cost value $E(s)$. Functions mapping other functions to real values are called *functionals* [72]. We seek a speed function s that minimizes the functional E under the constraints that the maximum speed is never exceeded and that all jobs are finished. Since we also need a scheduling policy (to decide which job to run when), we actually search n speed functions $s_j\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ telling us when and how to run j. The set of candidate functions is

$$\mathcal{S}_j \coloneqq \{\, f\colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0} \mid f \in C_{pr} \land \forall x \notin [r_j, d_j)\colon f(x) = 0 \,\}. \tag{3.1}$$

Let $\mathcal{S} \coloneqq \prod_{j\in J} \mathcal{S}_j$. To improve readability, we slightly abuse notation by using s for an element of $\mathcal{S}$ (that is a vector of the n different s_j's) as well as for the speed function of the schedule, i.e., the sum of the n different s_j's. We can formulate our optimization problem as the (infinite) mathematical program (SP) shown below.

$$\begin{aligned} \min_{s\in\mathcal{S}} \quad & E(\textstyle\sum_{j\in J} s_j) \\ \text{s.t.} \quad & \textstyle\sum_{j\in J} s_j(t) \leq s_{\max}(t) && \forall t \geq 0 && (3.2) \\ & \textstyle\int_{r_j}^{d_j} s_j(t)\,\mathrm{d}t \geq w_j && \forall j \in J && (3.3) \end{aligned}$$

An optimal solution minimizes the energy costs for the speed function $\sum_{j\in J} s_j$ without exceeding the maximal speed (Constraint (3.2)) and finishes all jobs (Constraint (3.3)). Note that we do not require the s_j to have pairwise disjoint supports.

In other words, the resulting schedule might run two jobs at the same time. Omitting this requirement is without loss of generality, as we can show how to transform such schedules to obtain pairwise disjoint supports.

Lemma 3.3.2 (EDF Schedule)**.** *Consider an arbitrary feasible solution $s \in \mathcal{S}$ to the optimization problem (*SP*). Then there exists a feasible solution $s' \in \mathcal{S}$ with $E(s') \le E(s)$ and the property $\forall j_1, j_2 \in J, t \in \mathbb{R}_{\ge 0} \colon s'_{j_1}(t) \cdot s'_{j_2}(t) > 0 \implies j_1 = j_2$.*

Proof. We transform solution s to a solution s' by employing *Earliest Deadline First* (EDF) scheduling. Intuitively, at every timepoint we run only the task that has the earliest deadline among all available tasks.

Let the indices of the jobs be ordered according to their deadline and assume, w.l.o.g., that no two jobs share the same deadline. We first transform s to a solution $\hat{s}$ such that for every job j, $\hat{s}$ processes exactly w_j workload (we call such schedules *non-wasting*). To do this we identify for each job j the earliest timepoint t_j such that $\int_{r_j}^{t_j} s_j(t)\,\mathrm{d}t = w_j$. Note that by constraint (3.3) we have that $t_j \le d_j$. We then set $\hat{s}_j(t) = 0$ for all $t \ge t_j$ and $\hat{s}_j(t) = s_j(t)$ for all $t < t_j$. Note that $E(\hat{s}) \le E(s)$ holds.

We next transform $\hat{s}$ into s'. To this end, set $t^* := \min_j r_j$ and let $\hat{w}_j := w_j$ denote the *remaining workload* of j, for all $j \in J$. We identify the job i that has the earliest deadline among all jobs j released by time t^* and with nonzero $\hat{w}_j$. Now, set t_i such that $\int_{t^*}^{t_i} \sum_j s_j(t)\,\mathrm{d}t = \hat{w}_i$. That is, the point at which i would be finished if it were exclusively processed by schedule s. If there is no release time in $[t^*, t_i)$, we set $s'_i(t) := \sum_j s_j(t)$ for every $t \in [t^*, t_i)$ and $s'_j(t) := 0$ for all other jobs. Then t^* is updated to t_i and $\hat{w}_i$ to 0. Otherwise, let r_k be the earliest release time in $[t^*, t_i)$. We set $s'_i(t) := \sum_j s_j(t)$ for every $t \in [t^*, r_k)$ and $s'_j(t) = 0$ for all other jobs. Finally we update $\hat{w}_i$ to $\hat{w}_i - \int_{t^*}^{r_k} s_i(t)\,\mathrm{d}t$ and t^* to r_k. We repeat the above step until all $\hat{w}_j$'s are set to 0.

The above transformation terminates, because in each iteration (new t^*) we make progress: either we move to the next release time, or one of the $\hat{w}_j$'s is set to zero. Also note that $E(\hat{s}) = E(s')$. This immediately follows by the fact that by the way the transformation is defined: for any t, $\sum_j \hat{s}_j(t) = \sum_j s'_j(t)$ holds. Further, since at every timepoint, t there exists at most one j such that $s'_j(t) > 0$, s' satisfies the property stated in the lemma. However, it is not immediately obvious that s' is feasible. We continue to show this. By the above transformation, we have for any j that $s'_j(t) = 0$ for all $t \in [r_{min}, r_j)$. It remains to show that the total workload of each job is processed before its deadline. More formally, we must have $\int_0^{d_j} s'_j(t)\,\mathrm{d}t = w_j$ for all jobs j. (The fact that $s'_j(t) = 0$ for $t \in (d_j, d_{max})$ then follows by the definition of the transformation). To this end, define for any job j, any timepoint t, and any solution s the value $F(t, j, s) := \int_0^t \sum_{i \le j} s_i(x)\,\mathrm{d}x$. Intuitively, $F(t, j, s)$ denotes the total workload of jobs with a deadline of at most d_j that s has finished by timepoint t. By Constraint (3.3) of (SP) and the first part of the transformation we have $F(d_j, j, \hat{s}) = \sum_{i=1}^{j} w_i$ for any job j.

We now show that for any j and t, we have the inequality $F(t, j, s') \ge F(t, j, \hat{s})$.

That is, s' finishes as least as much workload of jobs with deadline at most d_j by time t as $\hat{s}$. Indeed, assume that this is not the case and let t' be the first time such that $F(t', j, s') < F(t', j, \hat{s})$. In combination with the fact that s' satisfies the property stated in the lemma, this implies that at t' we have $\sum_{i=1}^{j} \hat{s}_i(t') > 0$ while $\sum_{i=1}^{j} s'_i(t') = 0$. However, by the definition of s', $\sum_{i=1}^{j} s'_i(t') = 0$ can only hold when all jobs with a release time $\leq t'$ and a deadline $\leq d_j$ are fully processed. This contradicts the assumption $F(t', j, s') < F(t', j, \hat{s})$, since $\hat{s}$ processes exactly w_j units for each job j. Thus, we must have $F(t, j, s') \geq F(t, j, \hat{s})$ for all t and j and, in particular, $F(d_j, j, s') \geq F(d_j, j, \hat{s})$ for all j. The lemma follows since $F(d_j, j, \hat{s}) = \sum_{i=1}^{j} w_i$. □

3.3.2. Characterizing Optimal Solutions

In the following, we formulate a more general optimization problem and derive (rather abstract) optimality conditions that can be viewed as an extended version of the KKT conditions. We will see in Section 3.3.3 how to apply these conditions to the ContBERS problem.

Let $N, m, n \in \mathbb{N}$. We consider an optimization problem for functionals over the set $\mathcal{F} := \prod_{j=1}^{N} \mathcal{F}_j$ and $m+n$ constraints, with $\mathcal{F}_j := \{\, g\colon \mathbb{R} \to \mathbb{R} \mid g \in C_{pr} \wedge \forall x \notin I_j\colon g(x) = 0 \,\}$ for some interval I_j. The j-th component of $f \in \mathcal{F}$ therefore is a right-continuous function g with finitely many discontinuities, and $g|_{\mathbb{R}\setminus I_j} = 0$. We also view the vectors $f \in \mathcal{F}$ as vector-valued functions $f\colon \mathbb{R} \to \mathbb{R}^N$.

We have an *objective function* $L\colon \mathbb{R} \times \mathbb{R}^N \to \mathbb{R}$ as well as two types of *constraint functions* $G_k, H_l\colon \mathbb{R} \times \mathbb{R}^N \to \mathbb{R}$ for $k \in \{\, 1, 2, \ldots, m \,\}$ and $l \in \{\, 1, 2, \ldots, n \,\}$. All these functions are assumed to be piecewise differentiable and convex in their second argument. For example, $G_k(x, y)$ with $x \in \mathbb{R}$ and $y \in \mathbb{R}^N$ is piecewise continuously differentiable and convex in y. We write ∇L (and similar for the other functions) to refer to the gradient of L taken with respect to the components of the second argument $y \in \mathbb{R}^N$ and $\nabla_j L$ for the j-th component of L's gradient. Let I be any interval in $\mathbb{R}$. The considered general optimization problem (GP) is

$$\begin{aligned}
\min_{f \in \mathcal{F}} \quad & \int_I L(x, f(x))\,\mathrm{d}x \\
\text{s.t.} \quad & G_k(x, f(x)) \leq 0 \qquad \forall x \in I, k \in \{\, 1, 2, \ldots, m \,\} \qquad \text{(I)} \\
& \int_I H_l(x, f(x))\,\mathrm{d}x \leq 0 \qquad \forall l \in \{\, 1, 2, \ldots, n \,\} \qquad \text{(II)}
\end{aligned}$$

Here, constraints of type (I) represent local constraints that hold at any point in time t (e.g., restricted processor speed). Constraints of type (II) represent global constraints that hold for some kind of volume (e.g., finished workload of a job). The following theorem provides sufficient optimality conditions for solutions of (GP).

Theorem 3.3.3 (Extended KKT conditions)**.** *Assume that $f \in \mathcal{F}$ is a feasible solution for (*GP*) with finite solution value. Furthermore, assume that there exist*

functions $\lambda_k \colon I \to \mathbb{R}_{\geq 0}$, $\lambda_k \in C_{pr}$, *and constants* $\mu_l \in \mathbb{R}_{\geq 0}$ *such that the following properties hold:*

1. *For all* $j \in \{1, 2, \ldots, N\}$ *and* $x \in I_j$, *we have*

$$\nabla_j L(x, f(x)) + \sum_{k=1}^{m} \lambda_k(x) \cdot \nabla_j G_k(x, f(x)) + \sum_{l=1}^{n} \mu_l \cdot \nabla_j H_l(x, f(x)) = 0. \quad (3.4)$$

2. *For all* $k \in \{1, 2, \ldots, m\}$ *and* $x \in I$, *we have* $\lambda_k(x) \cdot G_k(x, f(x)) = 0$.
3. *For all* $l \in \{1, 2, \ldots, n\}$, *we have* $\mu_l \cdot \int_I H_l(x, f(x))\,\mathrm{d}x = 0$.

Then f *is an optimal solution to (*GP*).*

Similar conditions have been used before (see for example [65]) but to the best of our knowledge not in this explicit form (as sufficient conditions). The rest of this subsection gives a (mostly self-contained) proof of Theorem 3.3.3.

Getting Rid of the Constraints. Using *Lagrange multipliers* $\lambda_k \colon I \to \mathbb{R}_{\geq 0}$, $\lambda_k \in C_{pr}$, $k \in \{1, 2, \ldots, m\}$, and $\mu_l \in \mathbb{R}_{\geq 0}$, $l \in \{1, 2, \ldots, n\}$, we can define the functional

$$\Lambda(f, \lambda, \mu) := \sum_{k=1}^{m} \int_I \lambda_k(x) \cdot G_k(x, f(x))\,\mathrm{d}x + \sum_{l=1}^{n} \mu_l \cdot \int_I H_l(x, f(x))\,\mathrm{d}x, \quad (3.5)$$

where $\lambda = (\lambda_1, \ldots, \lambda_m)$ and $\mu = (\mu_1, \ldots, \mu_n)$. This is the so called *Lagrangian*. By construction, we have $\Lambda \leq 0$ if f satisfies the constraints of (GP) (independently of λ and μ). This can be used to prove the following result known from duality theory [29, 72]:

Lemma 3.3.4. *Fix* λ *and* μ, *and consider an optimal solution* $\tilde{f} \in \mathcal{F}$ *to the minimization problem (*LGR*) given as*

$$\min_{f \in \mathcal{F}} D(f), \text{ where } D(f) := \int_I L(x, f(x))\,\mathrm{d}x + \Lambda(f, \lambda, \mu). \quad (3.6)$$

Assume that λ, μ, *and* $\tilde{f}$ *satisfy properties 2 and 3 of Theorem 3.3.3. If, additionally,* $\tilde{f}$ *fulfills the constraints of (*GP*), then* $\tilde{f}$ *is an optimal solution to (*GP*).*

Proof. For such λ, μ, and $\tilde{f}$, we have $\Lambda(\tilde{f}, \lambda, \mu) = 0$. Thus, when comparing $\tilde{f}$ to an arbitrary feasible solution f of (GP), we get

$$\begin{aligned}\int_I L\left(x, \tilde{f}(x)\right)\mathrm{d}x &= \int_I L\left(x, \tilde{f}(x)\right)\mathrm{d}x + \Lambda\left(\tilde{f}, \lambda, \mu\right) \leq \int_I L\left(x, f(x)\right)\mathrm{d}x + \Lambda\left(f, \lambda, \mu\right) \\ &\leq \int_I L\left(x, f(x)\right)\mathrm{d}x.\end{aligned}$$

The last inequality holds because f is a feasible solution to (GP), which implies $\Lambda(f, \lambda, \mu) \leq 0$. This proves the lemma's statement. □

Lemma 3.3.4 says that, in order to solve the minimization problem (GP) with its constraints, it is sufficient to solve (LGR) (which does not have constraints) for arbitrary, fixed dual variables, *but only if we can guarantee that the constraints are fulfilled.* This seems of small help, and in general such a solution might actually not exist. However, the dual variables give us an extra degree of freedom, and convexity ensures that our problem is “well-behaved”. Thus, our strategy is to find dual variables such that the optimal solution for (LGR) adheres to the constraints of (GP).

Convexity of (LGR). In order to solve (LGR), we first observe that the set $\mathcal{F}$ over which we optimize is convex. That is, for any two functions $f, g \in \mathcal{F}$ and $t \in [0,1]$ we have $(1-t) \cdot f + t \cdot g \in \mathcal{F}$. Another useful observation is that the objective D of (LGR) is convex (as a functional over $\mathcal{F}$). To see this, remember that L, G_k, and H_l are convex in their second argument. Moreover, D can be rewritten as

$$\begin{aligned} D(f) &= \int_I L\left(x, f(x)\right) \mathrm{d}x + \Lambda(f, \lambda, \mu) \\ &= \int_I \left(L(x, f(x)) + \sum_{k=1}^{m} \lambda_k(x) \cdot G_k(x, f(x)) + \sum_{l=1}^{n} \mu_l \cdot H_l(x, f(x)) \right) \mathrm{d}x \end{aligned}$$

$=: \int_I \tilde{L}(x, f(x), \lambda(x), \mu) \, \mathrm{d}x$, for a suitably defined $\tilde{L} \colon \mathbb{R} \times \mathbb{R}^N \times \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}$. The function $\tilde{L}$ is (as a positive sum of convex functions) convex in its second argument. Using monotonicity and linearity of the integration operator, we can prove the convexity of D:

$$\begin{aligned} &D\big(tf + (1-t)g\big) = \int_I \tilde{L}(x, tf(x) + (1-t)g(x), \lambda(x), \mu) \, \mathrm{d}x \\ &\leq \int_I \Big(t\tilde{L}(x, f(x), \lambda(x), \mu) + (1-t)\tilde{L}(x, g(x), \lambda(x), \mu) \Big) \, \mathrm{d}x \\ &= t \int_I \tilde{L}(x, f(x), \lambda(x), \mu) \, \mathrm{d}x + (1-t) \int_I \tilde{L}(x, g(x), \lambda(x), \mu) \, \mathrm{d}x = tD(f) + (1-t)D(g). \end{aligned}$$

Optimality Condition for (LGR). With D being convex, we can use the property that any local optimum is also globally optimal (cf. [55, Chap. 3]). Local optima can be characterized via their derivatives. Consider the (one-sided) directional derivatives $\delta_+ D(f, v) := \lim_{\varepsilon \to 0^+} \frac{D(f+\varepsilon v) - D(f)}{\varepsilon}$ of D at $f \in \mathcal{F}$ in any direction v with $f + v \in \mathcal{F}$. By convexity, a solution $f \in \mathcal{F}$ to (LGR) is (globally) optimal if and only if

$$\delta_+ D(f, v) \geq 0 \qquad \forall v \colon f + v \in \mathcal{F}. \tag{$\heartsuit$}$$

With this, we are now finally ready to prove Theorem 3.3.3.

Proof of Theorem 3.3.3. Assume we have functions $\lambda_k \colon I \to \mathbb{R}_{\geq 0}$ and constants $\mu_l \in \mathbb{R}_{\geq 0}$ as in Theorem 3.3.3, so that the properties 1 - 3 hold for a feasible solution

$f \in \mathcal{F}$ to (GP). Remember that L, G_k, and H_l are piecewise differentiable, implying that $\tilde{L}$ is piecewise differentiable. Similarly to the other functions, we write $\nabla\tilde{L}$ to denote $\tilde{L}$'s gradient taken with respect to the components of the second argument $y \in \mathbb{R}^N$. Then, by Leibniz and chain rule, we can write the directional derivative of Equation (♡) as

$$\begin{aligned} \delta_+ D(f, v) &= \frac{\mathrm{d}}{\mathrm{d}\varepsilon} \int_I \tilde{L}(x, f(x) + \varepsilon v(x), \lambda(x), \mu) \,\mathrm{d}x\big|_{\varepsilon=0} \\ &= \int_I \frac{\mathrm{d}}{\mathrm{d}\varepsilon} \tilde{L}(x, f(x) + \varepsilon v(x), \lambda(x), \mu)\big|_{\varepsilon=0} \,\mathrm{d}x = \int_I \Big\langle \nabla\tilde{L}(x, f(x), \lambda(x), \mu), v(x) \Big\rangle \,\mathrm{d}x \end{aligned} \tag{3.7}$$

Note that it may be necessary to split the integral for this operation as the involved functions are only *piecewise* differentiable. Now, by property 1 of Theorem 3.3.3, we have that component j of $\nabla\tilde{L}(x, f(x), \lambda(x), \mu)$ is equal to zero whenever $x \in I_j$. On the other hand, whenever $x \notin I_j$, we must have $v_j(x) = 0$ as otherwise $f_j(x) + v_j(x) \neq 0$, contradicting the fact that $f + v \in \mathcal{F}$. The integrand of Equation (3.7) thus vanishes, and $\delta_+ D(f, v) = 0$ for all directions v with $f + v \in \mathcal{F}$. This implies optimality of f for the optimization problem (LGR). As λ, μ, and f satisfy properties 2 and 3 of Theorem 3.3.3, we can apply Lemma 3.3.4 to show that f is an optimal solution of (GP). □

3.3.3. Extracting Structural Properties

We rewrite the mathematical program (SP) from Section 3.3.1 such that it has the form of the general mathematical program (GP) from Section 3.3.2. To this end, let T be the latest deadline and set $I := [0, T)$. We get the following convex problem:

$$\begin{aligned} \min_{s \in \mathcal{S}} \quad & E(s) \\ \text{s.t.} \quad & s(t) - s_{\max}(t) \leq 0 && \forall t \geq 0 && (3.8) \\ & \int_I \frac{w_j}{T} - s_j(t) \,\mathrm{d}t \leq 0 && \forall j \in J && (3.9) \\ & -s_j(t) \leq 0 && \forall j \in J, t \geq 0 && (3.10) \end{aligned}$$

Theorem 3.3.3 gives us a continuous version of the KKT conditions, which we can apply to extract a nice and combinatorial optimality condition for our problem. Note that the Constraints (3.8) and (3.10) translate to inequality constraints of type (I), whereas Constraint (3.9) corresponds to an inequality constraint of type (II).

Extended KKT Conditions for ContBERS. We now introduce a dual variable $\lambda\colon I \to \mathbb{R}_{\geq 0}$ for Constraint (3.8), dual variables $\mu_j \in \mathbb{R}_{\geq 0}$ for Constraint (3.9), and dual variables $\gamma_j\colon I \to \mathbb{R}_{\geq 0}$ for Constraint (3.10). Then the extended KKT conditions are:

1. Extended Stationarity: For all $j \in J$ and $t \in [r_j, d_j)$, the expression

$$\nabla_j \left(c(t)P\bigl(s(t)\bigr)\right) + \lambda(t) \cdot \nabla_j \bigl(s(t) - s_{\max}(t)\bigr) - \sum_{j \in J} \gamma_j(t) \cdot \nabla_j s_j(t) + \sum_{j \in J} \mu_j \cdot \nabla_j \left(\tfrac{w_j}{T} - s_j\left(t\right)\right)$$

   equals zero. Recall that ∇_j denotes the j-th component of the gradient of the *second* argument (where the arguments are t and $s(t)$) and, therefore, the partial derivative with respect to $s_j(t)$. Hence, the equality above is equivalent to

$$c(t)P'\bigl(s(t)\bigr) + \lambda(t) - \gamma_j(t) - \mu_j = 0. \tag{3.11}$$

2. Continuous complementary slackness conditions:

$$\lambda(t) \cdot \bigl(s(t) - s_{\max}(t)\bigr) = 0 \qquad \forall t \in I, \tag{3.12}$$

$$\gamma_j(t) \cdot s_j(t) = 0 \qquad \forall j \in J, t \in I. \tag{3.13}$$

3. Discrete complementary slackness conditions:

$$\mu_j \cdot \left(\int_I \frac{w_j}{T} - s_j(t) \,\mathrm{d}t\right) = 0 \qquad \forall j \in J. \tag{3.14}$$

Characterizing Optimality. We now use the above stated extended KKT conditions to characterize optimal solutions for our ContBERS scheduling problem. Using the jobs' release times and deadlines, we partition the time horizon into m non-overlapping, consecutive time intervals $T_i := [t_i, t_{i+1})$, $i \in \{1, 2, \dots, m\}$, where t_i is the i-th point in the set $\{r_j, d_j \mid j \in J\}$. Note that $m \leq 2n - 1$. We call T_i the i-th *atomic interval* and use $J(i) := \{j \in J \mid T_i \subseteq [r_j, d_j)\}$ to denote the set of jobs that are *active* in T_i (i.e., that may be scheduled in T_i).

The resulting scheduling condition is essentially a generalization of the well known optimality condition for the classical speed-scaling model from Yao et al. [79]. There, an important property of optimal schedules is that during the lifetime of a job j, speed never drops below the speed s_j used to process j. For our setting, we need a more general and complex optimality condition. In the following, we provide such a property (Definition 3.3.6) and prove that it characterizes optimal schedules (by proving Theorem 3.3.1) using our extended KKT conditions.

Definition 3.3.5 (Work-Transferable)**.** *For a given schedule and two atomic intervals T_i and $T_{i'}$, the* work-transferable *relation $i \to i'$ holds if there exists a job $j \in J(i) \cap J(i')$ with $\int_{t_i}^{t_{i+1}} s_j(t)\,\mathrm{d}t > 0$. Furthermore, let $\twoheadrightarrow$ be the reflexive transitive closure of $\to$.*

Definition 3.3.6 (Work-Balanced)**.** *We say that a schedule is* work-balanced *if there are constants $s_i \in \mathbb{R}$ for $i \in \{1, \dots, m\}$ so that 1. for any fixed atomic interval T_i the speed $s(t) \in \mathbb{R}$ at time $t \in [t_i, t_{i+1})$ is $\min(s_{\max}(t), c(t)^{-\frac{1}{\alpha-1}} \cdot s_i)$ and 2. for any two atomic intervals T_i and $T_{i'}$ with $i \twoheadrightarrow i'$, we have that $s_i \leq s_{i'}$.*

To get an intuition, assume $c(t)$ to be constant in each atomic interval. Then, the first property implies that, unless s_{max} forces us to run slower, we run at a constant speed in each atomic interval (which can be different for each atomic interval). The second property says that workload can only be transferred to intervals of higher speed (which would increase the cost). For non-constant $c(t)$, we have to weight the speed suitably.

In addition to the above properties, we call a schedule *non-wasting* if the workloads are exactly met (i.e., if for all $j \in J$ we have $w_j = \int_I s_j(t)\,\mathrm{d}t$). To ease the further discussion, we slightly abuse notation by extending the work-transferable relation "$\twoheadrightarrow$" to time points and jobs. More specifically, for an atomic interval T_i and a time $t \in \mathbb{R}_{\geq 0}$ we write $i \twoheadrightarrow t$ if for the atomic interval $T_{i'}$ with $t \in T_{i'}$ we have $i \twoheadrightarrow i'$. Similarly, for an atomic interval T_i and a job $j \in J$ we write $i \twoheadrightarrow j$ if there is an atomic interval $T_{i'}$ in which j is processed and for which $i \twoheadrightarrow i'$. Our analysis uses the following simple observations that follow immediately by these definitions:

Observation 3.3.7. *Consider a feasible schedule, an atomic interval i, a job j, and a time $t_0 \in [r_j, d_j)$. Then we have*

1. $\{\, i \mid i \twoheadrightarrow j \,\} \subseteq \{\, i \mid i \twoheadrightarrow t_0 \,\}$ *and*
2. *if* $s_j(t_0) > 0$, *then* $\{\, i \mid i \twoheadrightarrow t_0 \,\} = \{\, i \mid i \twoheadrightarrow j \,\}$.

We also need the following auxiliary lemma to characterize optimality via the above notions.

Lemma 3.3.8. *Assume that for a feasible schedule S there exist two timepoints $t_1 \in T_i$ and $t_2 \in T_{i'}$ such that*

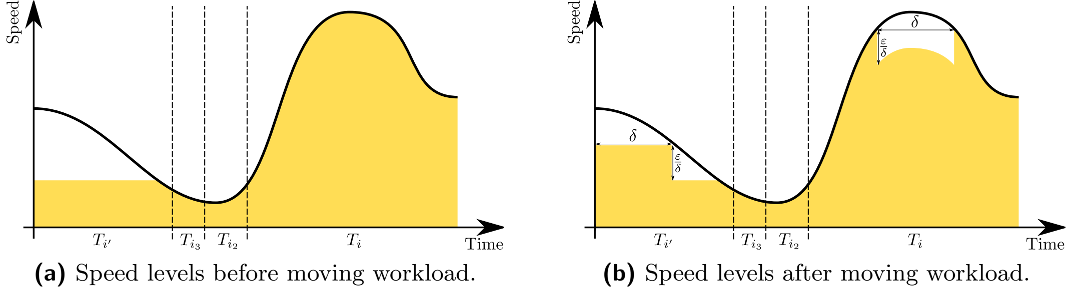
- $i \twoheadrightarrow i'$,
- $c(t_1)s(t_1)^{\alpha-1} > c(t_2)s(t_2)^{\alpha-1}$, *and*
- $s(t_2) < s_{max}(t_2)$.

Then S cannot be optimal.

Proof. To prove the lemma, we transform S to $\tilde{S}$, such that $\tilde{S}$ is feasible and $E(s) > E(\tilde{s})$. The transformation is as follows:

By the right-continuity of the involved functions, there exist intervals $I_i \coloneqq [t_1, t_1 + \varepsilon) \subseteq T_i$ and $I_{i'} \coloneqq [t_2, t_2 + \varepsilon) \subseteq T_{i'}$ for some $\varepsilon > 0$ such that $\min_{t \in I_i} c(t)s(t)^{\alpha-1} > \max_{t \in I_{i'}} c(t)s(t)^{\alpha-1}$ and $s(t') < s_{max}(t')$, for all $t' \in I_{i'}$.

By the definition of "$\twoheadrightarrow$", there exists a sequence of atomic intervals $T_i = T_{i_1}, T_{i_2}, \ldots, T_{i_l} = T_{i'}$, such that for each $y \in \{1, \ldots, l-1\}$ there holds $i_y \rightarrow i_{y+1}$. In other words, for every $y \in \{0, \ldots, l-1\}$, there exists a j_y such that $j_y \in J(i_y) \cap J(i_{y+1})$, and $\int_{t_{i_y}}^{t_{i_y+1}} s_{j_y}(t)\,\mathrm{d}t > 0$. Consecutively, for every such y we reduce the load of job j_y in the atomic interval T_{i_y} by $\delta > 0$, and increase the load of job j_y in $T_{i_{y+1}}$ by δ. At the same time we decrease the speed in I_i by δ/ε and increase the speed in $I_{i'}$ by δ/ε. It is easy to see that by choosing δ small enough, the resulting schedule $\tilde{S}$ is



(a) Speed levels before moving workload. **(b)** Speed levels after moving workload.

Figure 3.1.: Energy decrease when moving workload for the simplified problem with constant energy costs. The black line denotes the upper speed limit, whereas the level of the shaded area denotes the current speed level of the schedule.

feasible (although during the above procedure it may have been infeasible at times), and that $\min_{t \in I_i} c(t)\tilde{s}(t)^{\alpha-1} > \max_{t \in I_{i'}} c(t)\tilde{s}(t)^{\alpha-1}$ still holds. Further note that the cost only changes in the intervals I_i and $I_{i'}$.

Figure 3.1 visualizes the process described above for the simplified version with constant energy costs $c(t)$.

In I_i the energy cost decreases by:

$$
\begin{aligned}
\int_{I_i} c(t) \left(P(s(t)) - P(\tilde{s}(t))\right) \mathrm{d}t \\
\geq \int_{I_i} c(t) \left(\frac{\delta}{\varepsilon} P'(\tilde{s}(t))\right) \mathrm{d}t \\
\geq \int_{I_i} \frac{\delta}{\varepsilon} \min_{t \in I_i} \left(\alpha c(t)\tilde{s}(t)^{\alpha-1}\right) \mathrm{d}t \\
= \delta\alpha \cdot \min_{t \in I_i} \left(c(t)\tilde{s}(t)^{\alpha-1}\right),
\end{aligned}
$$

where the first inequality follows by the convexity of the power function.

On the other hand, by a similar calculation, the energy cost in $I_{i'}$ increases by at most:

$$
\delta\alpha \cdot \max_{t \in I_{i'}} \left(c(t)\tilde{s}(t)^{\alpha-1}\right).
$$

Since we chose δ so that $\min_{t \in I_i} \left(c(t)\tilde{s}(t)^{\alpha-1}\right) > \max_{t \in I_{i'}} \left(c(t)\tilde{s}(t)^{\alpha-1}\right)$ still holds, the lemma follows. □

We are now ready to prove our characterization of optimal schedules stated in Theorem 3.3.1.

Proof of Theorem 3.3.1. We start with the proof that being non-wasting and work-balanced is sufficient for optimality. Afterward, we show the necessity of both properties.

"⇐": Any feasible schedule S defines function variables s_j for a feasible solution. Here, $s_j(t)$ denotes the processing speed of job j at time t. In the following we set the dual variables and show that they satisfy the extended KKT conditions.

$$\lambda(t) := \sup_{t_0 \in T_k, k \twoheadrightarrow t} \big(c(t_0)P'\big(s(t_0)\big)\big) - c(t)P'\big(s(t)\big) \qquad \forall t \in [0, T)$$

$$\mu_j := \sup_{t_0 \in T_k, k \twoheadrightarrow j} \big(c(t_0)P'\big(s(t_0)\big)\big) \qquad \forall j \in J$$

$$\gamma_j(t) := \lambda(t) - \mu_j + c(t)P'\big(s(t)\big) \qquad \forall j \in J, t \in [r_j, d_j)$$

Moreover, we set $\gamma_j(t) := 0$ for $t \notin [r_j, d_j)$. We first need to show that these variables are dual-feasible (i.e., non-negative). We start with $\lambda(t)$. Consider the atomic interval T_i with $t \in T_i$. We obviously have $i \twoheadrightarrow t$, causing the supremum to consider t itself. Thus $\lambda(t)$ cannot be negative. The non-negativity of μ_j follows immediately from S being a feasible schedule. Because of this, there is an atomic interval T_i in which j is processed at some $t_0 \in T_i$ with speed $s(t_0) > 0$. For this atomic interval we have $i \twoheadrightarrow j$. Finally, the non-negativity of $\gamma_j(t)$ for any $j \in J$ and $t \in [r_j, d_j)$ follows immediately from Observation 3.3.71.

It remains to prove that the extended KKT conditions hold. The first condition, Equation (3.11), holds by definition of $\gamma_j(t)$. For Equation (3.13), fix $j \in J$, $t \in I$ and assume $s_j(t) > 0$. Then we must have $t \in [r_j, d_j)$. By applying Observation 3.3.72 we get $\{\, i \mid i \twoheadrightarrow t \,\} = \{\, i \mid i \twoheadrightarrow j \,\}$. This implies the equality of the supremum expressions in the definition of $\lambda(t)$ and μ_j and, thus, $\gamma_j(t) = 0$. Now look at Equation (3.12) for some fixed $t \geq 0$ with i such that $t \in T_i$ and assume $s(t) < s_{\max}(t)$. By definition of the work-balanced property, we must have $\sup_{t_0 \in T_i} \big(c(t_0)^{1/(\alpha-1)} s(t_0)\big) \leq s_i = c(t)^{1/(\alpha-1)} s(t)$. Moreover, any k with $k \twoheadrightarrow i$ satisfies $s_k \leq s_i$, which yields $\sup_{t_0 \in T_k} \big(c(t_0)^{1/(\alpha-1)} s(t_0)\big) \leq c(t)^{1/(\alpha-1)} s(t)$. By rearranging, we get $\sup_{t_0 \in T_k} \big(c(t_0) s(t_0)^{\alpha-1}\big) \leq c(t) s(t)^{\alpha-1}$. Since we have shown that $\lambda(t)$ cannot be negative, this yields $\lambda(t) = 0$. Finally, Equation (3.14) follows because S is non-wasting, which gives us $w_j = \int_{i \in I} s_j(t)\, \mathrm{d}t \ \forall j \in J$.

"⇒": First, we show that any optimal schedule S is work-balanced.

For every atomic interval T_ℓ, let $t_\ell := \arg\max_{t \in T_k, k \twoheadrightarrow \ell} s(t)c(t)^{\frac{1}{\alpha-1}}$, and $s_\ell := s(t_\ell)c(t_\ell)^{\frac{1}{\alpha-1}}$. For the sake of contradiction, assume that these s_ℓ's do not satisfy property (a) of work-balanced schedules (i.e., there exists some interval T_ℓ and $t^* \in T_\ell$ so that $s(t^*) \neq \min(s_{max}(t^*), c(t^*)^{-\frac{1}{\alpha-1}} \cdot s_\ell))$. Then it must be the case that $s(t^*) < s_{max}(t^*)$, since $s(t^*) > s_{max}(t^*)$ would contradict the feasibility of S, and $s(t^*) = s_{max}(t^*)$ would imply $c(t^*)^{-\frac{1}{\alpha-1}} \cdot s_\ell < s(t^*)$ and thus contradict our choice of s_ℓ. Therefore we have $s(t^*) < s_{max}(t^*)$ and $c(t^*)^{-\frac{1}{\alpha-1}} \cdot s_\ell \geq s(t^*)$. In fact, even the strict inequality $c(t^*)^{-\frac{1}{\alpha-1}} \cdot s_\ell > s(t^*)$ must hold, since equality would contradict the definition of t^*. Hence, all the

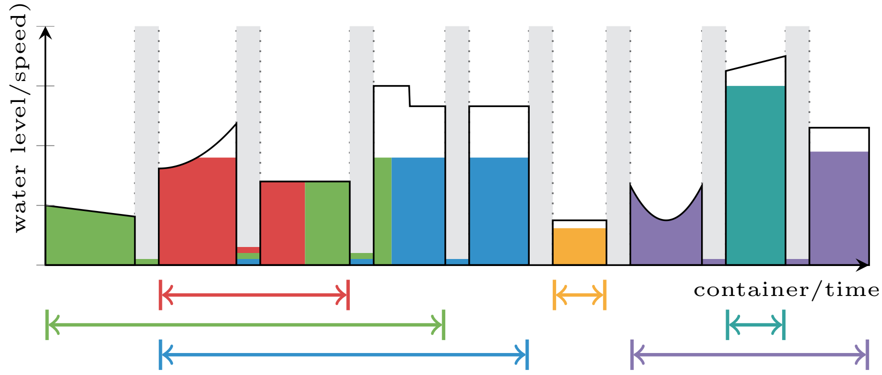


Figure 3.2.: Water-filling: Atomic interval containers whose upper borders represent the maximum speed function $s_{\max}$. This example is for constant energy costs and shows a work-balanced schedule.

properties of Lemma 3.3.8 are satisfied with $t_1 = t_\ell$ and $t_2 = t^*$, contradicting the optimality of S.

Property (b) of work-balanced schedules follows directly from the transitivity of $\twoheadrightarrow$ and the fact that s_i is defined as $\max_{t \in T_k, k \twoheadrightarrow i} s(t)c(t)^{\frac{1}{\alpha-1}}$.

Finally, assume S is optimal and not non-wasting. Obviously, we can uniformly decrease the speed for jobs with $w_j > \int_I s_j(t)\,\mathrm{d}t$ which leads to a lower energy cost and a contradiction.

□

3.4. Algorithm and Analysis

This section states our algorithm (Section 3.4.1) and proves both its correctness (via the work-balanced property; Theorem 3.4.2) and runtime bound (Theorem 3.4.3).

Overview. Our algorithm can be seen as pouring a liquid (workload of the jobs) into a number of connected containers (atomic intervals). The upper border of these containers is given by the maximum speed function $s_{\max}$, and neighboring containers are connected with valves. Pouring liquid into the containers causes the water levels to rise evenly among all non-full containers, while the valves ensure that the workload of a job does not leave its release-deadline interval. The process is stopped when all the liquid has been poured. The water level essentially corresponds to the speed used in the atomic interval. Figure 3.2 illustrates this intuition for constant energy costs.

If we consider dynamic electricity rates, the situation becomes more complicated. Here, the energy costs at time t can be interpreted as changing the liquids density over time. The water levels no longer correspond immediately to job speeds. Instead,

a job's speed at time t is essentially given by its water level times the density factor at time t. We note that water-filling is a natural way of viewing primal-dual algorithms (see, e.g., water-filling algorithms in [29, Chap. 5]).

3.4.1. Algorithm Description

Our algorithm works in rounds. In the first round, we find the set of consecutive atomic intervals $T_{i_1}, T_{i_1+1}, \ldots, T_{i_2}$ that require the "highest water level". This fixes the schedule for T_{i_1} to T_{i_2}. We then remove these atomic intervals and the scheduled jobs from the input, adapt the remaining jobs' release and deadlines, and start over again. We continue by formally defining *water levels* and describe more exactly how the algorithm computes a schedule in each round. See Listing 3.1 for the corresponding pseudocode.

Computing Water Levels. Consider a collection (union) I of atomic intervals. In the following, one can mostly think of I as a union of consecutive atomic intervals. However, for our proofs we also need to cover the case that I contains "holes" (i.e., a union of nonconsecutive atomic intervals). Define the set $J(I) \coloneqq \{\, j \in J \mid r_j \in I, d_j \in \overline{I} \,\}$ of jobs whose release times and deadlines are contained in I and the closure of I, respectively. Moreover, let $W(I) \coloneqq \sum_{j \in J(I)} w_j$ denote the total workload of these jobs. For a time $t \in \mathbb{R}_{\geq 0}$ let $\phi(t) \coloneqq c(t)^{-1/\alpha - 1}$ denote the *density factor at time t*. We define the *water level* $\rho(I) \in \mathbb{R}_{\geq 0}$ of I as the solution to the equation

$$W(I) = \int_I \min\bigl(\phi(t) \cdot \rho(I), s_{\max}(t)\bigr)\, \mathrm{d}t. \tag{3.15}$$

Equation (3.15) has a solution if and only if $W(I) \leq \int_I s_{\max}(t)\, \mathrm{d}t$. If this inequality is strict, the solution is unique. If it is an equality, we agree on $\rho(I) = \sup_{t \in I} \frac{s_{\max}(t)}{\phi(t)}$. If there is no solution to Equation (3.15), we define $\rho(I) \coloneqq \infty$.

Note that the computability of $\rho(I)$ depends not only on the ability to compute the involved integrals. Rather, one also must be able to solve an *integral equation* involving $s_{\max}$ and c. This is possible for practically relevant functions but can be nontrivial depending on $s_{\max}$ and c (e.g., for high-degree polynomials). In such cases, one can use numerical methods like binary search. Since our focus lies on the combinatorial scheduling aspect and continuity of the involved functions, we assume that $\rho(I)$ can be computed efficiently.

From Water Levels to Schedules. We describe the algorithm in an iterative way. This gives not the most efficient implementation but simplifies the analysis. Our algorithm iteratively computes a schedule for a subset of jobs and removes these from the input, creating a new subinstance of the original problem. This is then solved in the next iteration.

Set $I_0 \coloneqq \emptyset$ and consider an iteration $k \geq 1$. We first find indices i_{1k} and i_{2k} for which the water level $\rho_k \coloneqq \rho(I_k)$ of $I_k \coloneqq \bigl(\bigcup_{i=i_{1k}}^{i_{2k}} T_i\bigr) \setminus \bigl(\bigcup_{k' < k} I_{k'}\bigr)$ is maximal. If this

```
 1  A := { 1, 2, ..., m }    {remaining atomic interval indices}
 2  B := ∅, 𝒥 := J           {removed atomic interval indices and remaining jobs}
 3  while 𝒥 ≠ ∅:
 4      for each pair i_1 ≤ i_2 from A: compute water level ρ(i_1, i_2) := ρ(⋃_{i∈[i_1,i_2]\B} T_i)
 5      find maximum water level ρ_k := ρ(i_{1k}, i_{2k}) = max_{i_1,i_2} ρ(i_1, i_2)
 6      I_k := { i ∈ ℕ | i_{1k} ≤ i ≤ i_{2k}, i ∉ B }    {atomic intervals of this iteration}
 7      if ρ_k = ∞: return infeasible                    {feasibility check}
 8      set s_k(t) := min(φ(t) · ρ_k, s_max(t))          {speed to be used in atomic intervals of I_k}
 9      A := A \ I_k, B := B ∪ I_k, 𝒥 := 𝒥 \ J(I_k)
10      for all j ∈ 𝒥: update release times and deadlines
```

Listing 3.1: Primal-dual algorithm for the ContBERS problem. It returns the speed functions s_k to be used during the atomic intervals I_k of iteration k. To keep the pseudocode simple, we define the interval collections I_k as index sets instead of the actual unions of atomic intervals.

water level is ∞, the problem instance is infeasible. Otherwise, we can schedule all jobs $J_k := J(I_k)$ during I_k by using the EDF (earliest deadline first) scheduling policy and the speed function $s_k(t) := \min\bigl(\phi(t) \cdot \rho_k, s_{\max}(t)\bigr)$ during I_k. At the end of iteration k, we remove the scheduled jobs J_k and the time subset I_k from the input. This entails updating any remaining release time $r_j \in I_k$ to $\min\{\, t \geq r_j \mid t \notin \bigcup_{k' \leq k} I_{k'} \,\}$ and any remaining deadline $d_j \in \overline{I_k}$ to $\sup\{\, t \leq d_j \mid t \notin \bigcup_{k' \leq k} \overline{I_{k'}} \,\}$.

3.4.2. Correctness and Runtime

Before we state and prove our main result, we give an auxiliary lemma, showing that the water levels computed by our algorithm are monotonously decreasing.

Lemma 3.4.1. *The algorithm's water levels ρ_k are monotonously decreasing in k.*

Proof. Assume this is not true, so there is a minimal k such that $\rho_k < \rho_{k+1}$. First note that there are $u_1 < u_2$ and $v_1 < v_2$ with $I_k = [u_1, u_2) \setminus \bigcup_{k'<k} I_{k'}$ and $I_{k+1} = [v_1, v_2) \setminus \bigcup_{k'<k+1} I_{k'}$. We consider two cases:

Case 1: $[u_1, u_2] \cap [v_1, v_2] = \emptyset$

Note that, in this case, job removals and changes to release times or deadlines from iteration k cannot affect the job set $J(I_{k+1})$ in iteration $k+1$. But then, since our algorithm also considered I_{k+1} in iteration k, it would have computed the same water level $\rho_{k+1} > \rho_k$ for I_{k+1} in this iteration and chosen it instead of I_k. A contradiction.

Case 2: $[u_1, u_2] \cap [v_1, v_2] \neq \emptyset$

Consider the interval $I := I_k \cup I_{k+1}$. Because of $[u_1, u_2] \cap [v_1, v_2] \neq \emptyset$, our algorithm did consider I during iteration k. Moreover, note that $J(I)$ (in iteration k) contains both the job set $J(I_k)$ from iteration k and the job set $J(I_{k+1})$ from iteration $k+1$. Together with the definition of water levels, we

have

$$W(I) \geq W(I_k) + W(I_{k+1}) = \int_{I_k} \min\bigl(\phi(t) \cdot \rho_k, s_{\max}(t)\bigr)\,\mathrm{d}t + \int_{I_{k+1}} \min\bigl(\phi(t) \cdot \rho_{k+1}, s_{\max}(t)\bigr)\,\mathrm{d}t > \int_{I} \min\bigl(\phi(t) \cdot \rho_k, s_{\max}(t)\bigr)\,\mathrm{d}t.$$

Since the function $x \mapsto \int_I \min\bigl(\phi(t) \cdot x, s_{\max}(t)\bigr)\,\mathrm{d}t$ is nondecreasing, I's water level in iteration k must be larger than ρ_k, yielding a contradiction to our algorithm's choice.

□

Given this lemma, we can now prove the correctness of our algorithm.

Theorem 3.4.2 (Correctness)**.** *Consider an instance of the* ContBERS *problem. If there exists a feasible solution, our algorithm returns a work-balanced and non-wasting schedule. In particular, the returned schedule is optimal.*

Proof. Assume there is a feasible solution to the given instance. We first show that our algorithm returns a non-wasting schedule. Afterward, we show that this schedule is also work-balanced (implying its optimality by Theorem 3.3.1).

For the first iteration's water level, we have $\rho_1 < \infty$. Otherwise, even running all the time at maximal speed wouldn't finish all jobs in J_1, causing any schedule to be infeasible. Moreover, it is easy to see that EDF together with the speed function s_1 on I_1 yields a feasible (and non-wasting) schedule for the jobs J_1. This is because, by Equation (3.15), EDF with speed function s_1 exactly finishes the *workload* of all jobs within I_1 (i.e., when ignoring release times and deadlines). If this schedule is infeasible, there must be an $I'_1 \subset I_1$ with $W(I'_1) > \int_{I'_1} \min\bigl(\phi(t) \cdot \rho_1, s_{\max}(t)\bigr)\,\mathrm{d}t$. But then, since $x \mapsto \int_{I'_1} \min\bigl(\phi(t) \cdot x, s_{\max}(t)\bigr)\,\mathrm{d}t$ is nondecreasing and continuous, we get $\rho(I'_1) > \rho_1$, contradicting the maximality of ρ_1. Now consider a later iteration k and assume we found a feasible subschedule in the previous iteration $k-1$. We immediately get $\rho_k < \infty$ by Lemma 3.4.1 ($\rho_k = \infty$ would contradict $\rho_k \leq \rho_{k-1} < \infty$). The feasibility and non-wasting property of EDF with speed function s_k in I_k follows by the same argument as for the first iteration.

We continue to show that the algorithm computes a work-balanced schedule. To this end, we show that the constants s_i from Definition 3.3.6 are given by the water levels ρ_k with $T_i \subseteq I_k$. The first part of this definition is obviously met, as it corresponds exactly to our definition of water levels and the speed functions in I_k. For the second part, note that we need only to consider two atomic intervals $T_i \subseteq I_k$ and $T_{i'} \subseteq I_{k'}$ from *different iterations* $k < k'$ (if they are from the same iteration, their water levels match, such that the definition's second part holds trivially). By construction of the algorithm, we cannot have $i \twoheadrightarrow i'$: no j scheduled in T_i is active outside of $\bigcup_{k'' \leq k} I_{k''}$, and the same holds for any j scheduled in $\bigcup_{k'' \leq k} I_{k''}$ (and thus no such job is active in $I_{k'}$). On the other hand, if $i' \twoheadrightarrow i$, the second part of Definition 3.3.6 holds as $\rho_{k'} \leq \rho_k$ by Lemma 3.4.1. □

Note that the running time of any algorithm for the ContBERS problem inherently depends on the ability to perform advanced computations on continuous functions. Depending on $s_{\max}$ and the cost factor function c, relying on numerical methods might even be unavoidable. Since we are interested in the scheduling aspect of the model, the following runtime discussion assumes computations (like integrals, solving equations, taking the minimum etc.) involving continuous functions can be performed in constant time. As noted earlier, our iterative implementation is not the most efficient one, but it is convenient for our correctness analysis. A rather simple improvement can be achieved by precomputing the water levels for all pairs i_1 and i_2 of atomic intervals beforehand and merely updating these values at the end of each iteration. This immediately yields the same cubic running time as known from the original YDS algorithm [79][1]:

Theorem 3.4.3 (Runtime)**.** *The* ContBERS *problem can be solved in time* $\mathcal{O}(n^3)$*.*

[1]There are improved implementations of YDS with runtime $\mathcal{O}(n^2 \log n)$ [63] and $\mathcal{O}(n^2)$ [64].

Part II.

MAKESPAN SCHEDULING

4. Graph Balancing with Light Hyper Edges

4.1. Introduction

Let $\mathcal{J}$ be a set of n jobs and $\mathcal{M}$ a set of m machines. Each job $j \in \mathcal{J}$ has a *weight* w_j and can be assigned to a specific subset of the machines. An assignment $\sigma : \mathcal{J} \to \mathcal{M}$ is a mapping where each job is mapped to a machine to which it can be assigned. The objective is to minimize the *makespan*, defined as $\max_{i \in \mathcal{M}} \sum_{j:\sigma(j)=i} w_j$. This is the classical MAKESPAN MINIMIZATION IN RESTRICTED ASSIGNMENT ($R|p_{ij} \in \{p_j, \infty\}|C_{\max}$), itself a special case of the MAKESPAN MINIMIZATION IN UNRELATED MACHINES ($R||C_{\max}$), where a job j has possibly different weight w_{ij} on different machines $i \in \mathcal{M}$. In the following, we just call them RESTRICTED ASSIGNMENT and UNRELATED MACHINE PROBLEM for short.

The first constant approximation algorithm for both problems is given by Lenstra, Shmoys, and Tardos [58] in 1990, where the ratio is 2. They also show that RESTRICTED ASSIGNMENT (hence also the UNRELATED MACHINE PROBLEM) cannot be approximated within 1.5 unless P=NP, even if there are only two job weights. The upper bound of 2 and the lower bound of 1.5 have been essentially unimproved in the intervening 25 years. How to close the gap continues to be one of the central topics in approximation algorithms. The recent book of Williamson and Shmoys [78] lists this as one of the ten open problems.

Our Result

We consider a special case of RESTRICTED ASSIGNMENT, called GRAPH BALANCING WITH LIGHT HYPER EDGES, which is a generalization of the GRAPH BALANCING problem introduced by Ebenlendr, Krčál and Sgall [36]. There the restriction is that every job can be assigned to only two machines, and hence the problem can be interpreted in a graph-theoretic way: each machine is represented by a node, and each job is represented by an edge. The goal is to find an orientation of the edges so that the maximum weight sum of the edges oriented towards a node is minimized. In our problem, jobs are partitioned into *heavy* and *light*, and we assume that heavy jobs can go to only two machines while light jobs can go to any number of machines[1]. In the graph-theoretic interpretation, light jobs are represented by hyper edges, while heavy jobs are represented by regular edges.

[1] If some jobs can be assigned to just one machine, then it is the same as saying a machine has some *dedicated load*. All our algorithms can handle arbitrary dedicated loads on the machines.

We present approximation algorithms with performance guarantee *strictly better than* 2 in the following settings. For simplicity of presentation, we assume that all job weights w_j are integral (this assumption is just for ease of exposition and can be easily removed).

Two job sizes: Suppose that heavy jobs are of weight W and light jobs are of weight w, and $w < W$. We give a 1.5-approximation algorithm, matching the general lower bound of RESTRICTED ASSIGNMENT (it should be noted that this lower bound is established in an even more restrictive setting [11, 36], where all jobs can only go to two machines and there are only two different job weights). This is the first time the lower bound is matched in a nontrivial case of RESTRICTED ASSIGNMENT (without specific restrictions on the job weight values). In fact, sometimes our algorithm achieves an approximation ratio strictly better than 1.5. Supposing that $w \leq \frac{W}{2}$, the ratio we get is $1 + \frac{\lfloor W/2 \rfloor}{W}$.

Arbitrary job sizes: Suppose that $\beta \in [4/7, 1)$ and W is the largest given weight. A heavy job has weight in $(\beta W, W]$ while a light job has weight in $(0, \beta W]$. We give a $(5/3 + \beta/3)$-approximation algorithm.

Both algorithms have the running time of $\mathcal{O}\big(n^2 m^3 \log\big(\sum_{j \in \mathcal{J}} w_j\big)\big)$.[2]

The general message of our result is clear: as long as the heaviest jobs have only two choices, it is relatively easy to break the barrier of 2 in the upper bound of RESTRICTED ASSIGNMENT. This should coincide with our intuition. The heavy jobs are in a sense the "trouble-makers". A mistake on them causes bigger damage than a mistake on lighter jobs. Restricting the choices of the heavy jobs thus simplifies the task.

The original GRAPH BALANCING problem assumes that all jobs can be assigned to only two machines and the algorithm of Ebenlendr et al. [36] gives a 1.75-approximation. According to [70], their algorithm can be extended to our setting: given any $\beta \in [0.5, 1)$, they can obtain a $(3/2 + \beta/2)$-approximation. Although this ratio is superior to ours, let us emphasize two interesting aspects of our approach.

(1) The algorithm of Ebenlendr et al. requires solving a linear program (in fact, almost all known algorithms for the problem are LP-based), while our algorithms are purely combinatorial. In addition to the advantage of faster running time, our approach introduces new proof techniques (which do not involve linear programming duality).

(2) In GRAPH BALANCING, Ebenlendr et al. showed that with only two job weights and dedicated loads on the machines, their strongest LP has the integrality gap of 1.75, while we can break the gap. Our approach thus offers a possible angle to circumvent the barrier posed by the integrality gap, and has the potential of seeing

[2] For simplicity, here we upper bound $\sum_{j \in \mathcal{J}} a_j$, where a_j is the number of the machines j can be assigned to, by nm.

further improvement.

Before explaining our technique in more detail, we should point out another interesting connection with a result of Svensson [73] for general RESTRICTED ASSIGNMENT. He gave two local search algorithms, which terminate (but it is unknown whether in polynomial time) and (1) with two job weights $\{\epsilon, 1\}$, $0 < \epsilon < 1$, the returned solution has an approximation ratio of $5/3 + \epsilon$, and (2) with arbitrary job weights, the returned solution has an approximation ratio of ≈ 1.94. It is worth noting that his analysis is done via the primal-duality of the configuration-LP (thus integrality gaps smaller than two for the configuration-LP are implied). With two job weights, our algorithm has some striking similarity to his algorithm. We are able to prove our algorithm terminates in polynomial time—but our setting is more restrictive. A very interesting direction for future work is to investigate how the ideas in the two algorithms can be related and combined.

Our Technique

Our approach is inspired by that of Gairing et al. [39] for general RESTRICTED ASSIGNMENT. So let us first review their ideas. Suppose that a certain optimal makespan t is guessed. Their core algorithm either (1) correctly reports that t is an underestimate of OPT, or (2) returns an assignment with makespan at most $t + W - 1$. By a binary search on the smallest t for which an assignment with makespan $t+W-1$ is returned, and the simple fact that $\mathrm{OPT} \geq W$, they guarantee the approximation ratio of $\frac{t+W-1}{\mathrm{OPT}} \leq 1 + \frac{W-1}{\mathrm{OPT}} \leq 2 - \frac{1}{W}$ (the first inequality holds because t is the smallest number an assignment is returned by the core algorithm). Their core algorithm is a preflow-push algorithm. Initially all jobs are arbitrarily assigned. Their algorithm tries to redistribute the jobs from overloaded machines, i.e., those with load more than $t+W-1$, to those that are not. The redistribution is done by pushing the jobs around while updating the height labels (as commonly done in preflow-push algorithms). The critical thing is that after a polynomial number of steps, if there are still some overloaded machines, they use the height labels to argue that t is a wrong guess, i.e., $\mathrm{OPT} \geq t + 1$. Our contribution is a refined core algorithm in the same framework. With a guess t of the optimal makespan, our core algorithm either (1) correctly reports that $\mathrm{OPT} \geq t+1$, or (2) returns an assignment with makespan at most $(5/3 + \beta/3)t$.

We divide all jobs into two categories, the *rock jobs* $\mathbb{R}$, and the *pebble jobs* $\mathbb{P}$ (not to be confused with heavy and light jobs). The former consists of those with weights in $(\beta t, t]$ while the latter includes all the rest. We use the rock jobs to form a graph $G_{\mathbb{R}} = (V, \mathbb{R})$, and assign the pebbles arbitrarily to the nodes. Our core algorithm will push around the pebbles so as to redistribute them. Observe that as $t \geq W$, all rocks are heavy jobs. So the formed graph $G_{\mathbb{R}}$ has only simple edges (no hyper edges). As $\beta \geq 4/7$, if $\mathrm{OPT} \leq t$, then every node can receive at most one rock job in the optimal solution. In fact, it is easy to see that we can simply assume that the formed graph $G_{\mathbb{R}}$ is a disjoint set of trees and cycles. Our entire task boils down to the following:

> Redistribute the pebbles so that there exists an orientation of the edges in $G_{\mathbb{R}}$ in which each node has total load (from both rocks and pebbles) at most $(5/3 + \beta/3)t$; and if not possible, gather evidence that t is an underestimate.

Intuitively speaking, our algorithm maintains a certain *activated set* $\mathbb{A}$ of nodes. Initially, this set includes those nodes whose total loads of pebbles cause conflicts in the orientation of the edges in $G_{\mathbb{R}}$. A node "reachable" from a node in the activated set is also included into the set. (Node u is reachable from node v if a pebble in v can be assigned to u.) Our goal is to push the pebbles among nodes in $\mathbb{A}$, so as to remove all conflicts in the edge orientation. Either we are successful in doing so, or we argue that the total load of all pebbles currently owned by the activated set, together with the total load of the rock jobs assigned to $\mathbb{A}$ in any *feasible orientation* of the edges in $G_{\mathbb{R}}$ (an orientation in $G_{\mathbb{R}}$ is *feasible* if every node receives at most one rock), is strictly larger than $t \cdot |\mathbb{A}|$. The progress of our algorithm (hence its running time) is monitored by a potential function, which we show to be monotonically decreasing.

The most sophisticated part of our algorithm is the "activation strategy". We initially add nodes into $\mathbb{A}$ if they cause conflicts in the orientation or can be (transitively) reached from such. However, sometimes we also include nodes that do not fall into the two categories. This is purposely done for two reasons: pushing pebbles from these nodes may help alleviate the conflict in edge orientation indirectly; and their presence in $\mathbb{A}$ strengthens the contradiction proof.

Due to the intricacy of our main algorithm, we first present the algorithm for the two job weights case in Section 4.3 and then present the main algorithm for the arbitrary weights in Section 4.4. The former algorithm is significantly simpler (with a straightforward activation strategy) and contains many ingredients of the ideas behind the main algorithm.

Related Work

For RESTRICTED ASSIGNMENT, besides the several recent advances mentioned earlier, see the survey of Leung and Li for other special cases [59]. For two job weights, Chakrabarti, Khanna and Li [31] showed that using the configuration-LP, they can obtain a $(2-\delta)$-approximation for a fixed $\delta > 0$ (and note that there is no restriction on the number of machines a job can go to). Kolliopoulos and Moysoglou [54] also considered the two job weights case. In the GRAPH BALANCING setting (with two job weights), they gave a 1.652-approximation algorithm using a flow technique (thus they also break the integrality gap in [37]). They also showed that the optimal makespan for RESTRICTED ASSIGNMENT with two job weights can be estimated in polynomial time within a factor of at most 1.883 (and this is further improved to 1.833 in [31]).

For UNRELATED MACHINES, Shchepin and Vakhania [71] improved the approximation ratio to $2 - 1/m$. A combinatorial 2-approximation algorithm was given by Gairing, Monien, and Woclaw [40]. Verschae and Wiese [76] showed that the

configuration-LP has integrality gap of 2, even if every job can be assigned to only two machines. They also showed that it is possible to achieve approximation ratios strictly better than 2 if the job weights w_{ij} respect some constraints.

4.2. Preliminaries

Let t be a guess of OPT. Given t, our two core algorithms either report that OPT $\geq t+1$, or return an assignment with makespan at most $1.5t$ or $(5/3+\beta/3)t$, respectively. We conduct a binary search on the smallest $t \in [W, \sum_{j \in \mathcal{J}} w_j]$ for which an assignment is returned by the core algorithms. This particular assignment is then the desired solution.

We now explain the initial setup of the core algorithms. In our discussion, we will not distinguish a machine and a node. Let $dl(v)$ be the dedicated load of v, i.e., the sum of the weights of jobs that can only be assigned to v. We can assume that $dl(v) \leq t$ for all nodes v. Let $\mathcal{J}' \subseteq \mathcal{J}$ be the jobs that can be assigned to at least two machines. We divide $\mathcal{J}'$ into rocks $\mathbb{R}$ and pebbles $\mathbb{P}$. A job $j \in \mathcal{J}'$ is a rock,

- in the 2 job weights case (Section 4.3), if $w_j > t/2$ and $w_j = W$;
- in the general job weights case (Section 4.4), if $w_j > \beta t$.

A job $j \in \mathcal{J}'$ that is not a rock is a pebble. Define the graph $G_{\mathbb{R}} = (V, \mathbb{R})$ as a graph with machines $\mathcal{M}$ as node set and rocks $\mathbb{R}$ as edge set. By our definition, a rock can be assigned to exactly two machines. So $G_{\mathbb{R}}$ has only simple edges (no hyper edges). For the sake of convenience, we call the rocks just "edges", avoiding ambiguity by exclusively using the term "pebble" for the pebbles.

Suppose that OPT $\leq t$. Then a machine can receive at most one rock in the optimal solution. If any connected component in $G_{\mathbb{R}}$ has more than one cycle, we can immediately declare that OPT $\geq t+1$. If a connected component in $G_{\mathbb{R}}$ has exactly one cycle, we can direct all edges away from the cycle and remove these edges, i.e., assign the rock to the node v to which it is directed. W.L.O.G, we can assume that this rock is part of v's dedicated load. (Also observe that then node v must become an isolated node). Finally, we can eliminate cycles of length 2 in $G_{\mathbb{R}}$ with the following simple reduction. If a pair of nodes u and v is connected by two distinct rocks $r1$ and $r2$, remove the two rocks, add $\min(w_{r1}, w_{r2})$ to both u's and v's dedicated load, and introduce a new pebble of weight $|w_{r1} - w_{r2}|$ between u and v. Let Ψ denote the set of orientations in $G_{\mathbb{R}}$ where each node has at most one incoming edge. We use a proposition to summarize the above discussion.

Proposition 4.2.1. *We can assume that*

- *the rocks in $\mathbb{R}$ correspond to the edge set of the graph $G_{\mathbb{R}}$, and all pebbles can be assigned to at least two machines;*
- *the graph $G_{\mathbb{R}}$ consists of disjoint trees, cycles (of length more than 2), and isolated nodes;*

- *for each node $v \in V$, $dl(v) \leq t$;*
- *if* OPT $\leq t$, *then the orientation of the edges in $G_{\mathbb{R}}$ in the optimal assignment must be one of those in Ψ.*

4.3. The 2-Valued Case

In this section, we describe the core algorithm for the two job weights case, with the guessed makespan $t \geq W$. Observe that when $t \in [W, 2w)$, if OPT $\leq t$, then every node can receive at most one job (pebble or rock) in the optimal assignment. Hence, we can solve the problem exactly using the standard max-flow technique. So in the following, assume that $t \geq 2w$. Furthermore, let us first assume that $t < 2W$ (the case of $t \geq 2W$ will be discussed at the end of the section). Then the rocks have weight W and the pebbles have weight w. Initially, the pebbles are arbitrarily assigned to the nodes. Let $pl(v)$ be the total weight of the pebbles assigned to node v.

Definition 4.3.1. *A node v is*

- uncritical, *if $dl(v) + pl(v) \leq 1.5t - W - w$;*
- critical, *if $dl(v) + pl(v) > 1.5t - W$;*
- hypercritical, *if $dl(v) + pl(v) > 1.5t$.*

(Notice that it is possible that a node is neither uncritical nor critical.)

Definition 4.3.2. *Each tree, cycle, or isolated node in $G_{\mathbb{R}}$ is a* system. *A system is* bad *if any of the following conditions holds.*

- *It is a tree and has at least two critical nodes, or*
- *It is a cycle and has at least one critical node, or*
- *It contains a hypercritical node.*

A system that is not bad is good.

If all systems are good, then orienting the edges in each system such that every node has at most one incoming edge gives us a solution with makespan at most $1.5t$. So let assume that there is at least one bad system.

We next define the *activated set* $\mathbb{A}$ of nodes constructively. Roughly speaking, we will move pebbles around the nodes in $\mathbb{A}$ so that either there is no more bad system left, or we argue that, in every feasible assignment, *some* nodes in $\mathbb{A}$ cannot handle their total loads, thereby arriving at a contradiction.

In the following, if a pebble in u can be assigned to node v, we say v is reachable from u. Node v is reachable from $\mathbb{A}$ if v is reachable from any node $u \in \mathbb{A}$. A node added into $\mathbb{A}$ is *activated*.

Informally, all nodes that cause a system to be bad are activated. A node reachable from $\mathbb{A}$ is also activated. Furthermore, suppose that a system is good and it has a critical node v (thus the system cannot be a cycle). If any other node u in the same system is activated, then so is v. We now give the formal procedure EXPLORE1 in Figure 4.1. Notice that in the process of activating the nodes, we also define their *levels*, which will be used later for the algorithm and the potential function.

EXPLORE1

Initialize $\mathbb{A} := \{v | v \text{ is hypercritical, or } v \text{ is critical in a bad system}\}$.
  Set LEVEL(v) := 0 for all nodes in $\mathbb{A}$; $i := 0$.

While $\exists v \notin \mathbb{A}$ reachable from $\mathbb{A}$ **do:**
  $i := i + 1$.
  $\mathbb{A}_i := \{v \notin \mathbb{A} | v \text{ reachable from } \mathbb{A}\}$.
  $\mathbb{A}'_i := \{v \notin \mathbb{A} |\ v \text{ is critical in a good system and } \exists u \in \mathbb{A}_i \text{ in the same system}\}$.
  Set LEVEL(v) := i for all nodes in $\mathbb{A}_i$ and $\mathbb{A}'_i$.
  $\mathbb{A} := \mathbb{A} \cup \mathbb{A}_i \cup \mathbb{A}'_i$.
For each node $v \notin \mathbb{A}$, set LEVEL(v) = ∞.

Figure 4.1.: The procedure EXPLORE1.

The next proposition follows straightforwardly from EXPLORE1.

Proposition 4.3.3. *The following holds.*

1. *All nodes reachable from* $\mathbb{A}$ *are in* $\mathbb{A}$.
2. *Suppose that* v *is reachable from* $u \in \mathbb{A}$. *Then* LEVEL(v) $\leq$ LEVEL(u) + 1.
3. *If a node* v *is critical and there exists another node* $v' \in \mathbb{A}$ *in the same system, then* LEVEL(v) $\leq$ LEVEL(v').
4. *Suppose that node* $v \in \mathbb{A}$ *has* LEVEL(v) $= i > 0$. *Then there exists another node* $u \in \mathbb{A}$ *with* LEVEL(u) $= i - 1$ *so that either* v *is reachable from* u, *or there exists another node* $v' \in \mathbb{A}$ *reachable from* u *with* LEVEL(v') $= i$ *in the same system as* v *and* v *is critical.*

After EXPLORE1, we apply the PUSH operation (if possible), defined as follows.

Definition 4.3.4. PUSH *operation: push a pebble from* u^* *to* v^* *if the following conditions hold.*

1. *The pebble is at* u^* *and it can be assigned to* v^*.
2. LEVEL(v^*) = LEVEL(u^*) + 1.

3. *v^* is uncritical, or v^* is in a good system that remains good with an additional weight of w at v^*.*

4. *Subject to the above three conditions, choose a node u^* so that* LEVEL(u^*) *is minimized (if there are multiple candidates, pick any).*

Our algorithm can be simply described as follows.

> **Algorithm 1**: As long as there is a bad system, apply EXPLORE1 and PUSH operation repeatedly. When there is no bad system left, return a solution with makespan at most $1.5t$. If at some point, PUSH is no longer possible, declare that $\text{OPT} \geq t+1$.

Lemma 4.3.5. *When there is at least one bad system and the* PUSH *operation is no longer possible,* $\text{OPT} \geq t+1$.

Proof. Let $\mathbb{A}(S)$ denote the set of activated nodes in system S. Recall that Ψ denotes the set of all orientations in $G_{\mathbb{R}}$ in which each node has at most one incoming edge. We prove the lemma via the following claim.

Claim 4.3.6. *Let S be a system.*

- *Suppose that S is bad. Then*

$$W\cdot(\min_{\psi\in\Psi} \textit{number of rocks to } \mathbb{A}(S) \textit{ according to } \psi)+\sum_{v\in\mathbb{A}(S)} pl(v)+dl(v) > |\mathbb{A}(S)|t. \tag{4.1}$$

- *Suppose that S is good. Then*

$$W\cdot(\min_{\psi\in\Psi} \textit{number of rocks to } \mathbb{A}(S) \textit{ according to } \psi)+\sum_{v\in\mathbb{A}(S)} pl(v)+dl(v) > |\mathbb{A}(S)|t-w. \tag{4.2}$$

Observe that the term $|\mathbb{A}(S)|t$ is the maximum total weight that all nodes in $\mathbb{A}(S)$ can handle if $\text{OPT} \leq t$. As pebbles owned by nodes in $\mathbb{A}$ can only be assigned to the nodes in $\mathbb{A}$, by the pigeonhole principle, in all orientations $\psi \in \Psi$, and all possible assignments of the pebbles, at least one bad system S has at least the same number of pebbles in $\mathbb{A}(S)$ as the current assignment, or a good system S has at least one more pebble than it currently has in $\mathbb{A}(S)$. In both cases, we reach a contradiction. □

Proof of Claim 4.3.6: First observe that in all orientations in Ψ, the nodes in $\mathbb{A}(S)$ have to receive at least $|\mathbb{A}(S)|-1$ rocks. If S is a cycle, then the nodes in $\mathbb{A}(S)$ have to receive exactly $|\mathbb{A}(S)|$ rocks.

Next observe that none of the nodes in $\mathbb{A}(S)$ is uncritical, since otherwise, by Proposition 4.3.3(4) and Definition 4.3.4(3), the PUSH operation would still be possible. By the same reasoning, if S is a tree and $\mathbb{A}(S) \neq \emptyset$, at least one node $v \in \mathbb{A}(S)$

is critical; furthermore, if $|\mathbb{A}(S)| = 1$, this node v satisfies $dl(v)+pl(v) > 1.5t-w$, as an additional weight of w would make v hypercritical. Similarly, if S is an isolated node $v \in \mathbb{A}$, then $dl(v) + pl(v) > 1.5t - w$.

We now prove the claim by the following case analysis.

1. Suppose that S is a good system and $\mathbb{A}(S) \neq \emptyset$. Then either S is a tree and $\mathbb{A}(S)$ contains exactly one critical (but not hypercritical) node, or S is an isolated node, or S is a cycle and has no critical node. In the first case, if $|\mathbb{A}(S)| \geq 2$, the LHS of (4.2) is at least

$$(1.5t - W + 1) + (|\mathbb{A}(S)| - 1)(1.5t - W - w + 1) + (|\mathbb{A}(S)| - 1)W =$$
$$|\mathbb{A}(S)|t + (|\mathbb{A}(S)| - 2)(0.5t - w + 1) + t - W - w + 2 > |\mathbb{A}(S)|t - w,$$

   using the fact that $0.5t \geq w$, $t \geq W$, and $|\mathbb{A}(S)| \geq 2$. If, on the other hand, $|\mathbb{A}(S)| = 1$, then the LHS of (4.2) is strictly more than

$$1.5t - w \geq t = |\mathbb{A}(S)|t,$$

   and the same also holds for the case when S is an isolated node. Finally, in the third case, the LHS of (4.2) is at least

$$|\mathbb{A}(S)|(1.5t - W - w + 1) + |\mathbb{A}(S)|W > |\mathbb{A}(S)|t.$$

2. Suppose that $\mathbb{A}(S)$ contains at least two critical nodes, or that S is a cycle and $\mathbb{A}(S)$ has at least one critical node. In both cases, S is a bad system. Furthermore, the LHS of (4.1) can be lower-bounded by the same calculation as in the previous case with an extra term of w.

3. Suppose that $\mathbb{A}(S)$ contains a hypercritical node. Then the system S is bad, and the LHS of (4.1) is at least

$$(1.5t + 1) + (|\mathbb{A}(S)| - 1)(1.5t - W - w + 1) + (|\mathbb{A}(S)| - 1)W =$$
$$|\mathbb{A}(S)|t + (|\mathbb{A}(S)| - 1)(0.5t - w + 1) + 0.5t + 1 > |\mathbb{A}(S)|t,$$

   where the inequality holds because $0.5t \geq w$. □

We argue that Algorithm 1 terminates in polynomial time by the aid of a potential function, defined as

$$\Phi = \sum_{v \in \mathbb{A}} (|V| - \textsc{Level}(v)) \cdot (\text{number of pebbles at } v).$$

Trivially, $0 \leq \Phi \leq |V| \cdot |\mathbb{P}|$. The next lemma implies that Φ is monotonically decreasing after each Push operation.

Lemma 4.3.7. *For each node $v \in V$, let $\textsc{Level}(v)$ and $\textsc{Level}'(v)$ denote the levels before and after a Push operation, respectively. Then $\textsc{Level}'(v) \geq \textsc{Level}(v)$.*

Proof. We prove by contradiction. Suppose that there exist nodes x with $\textsc{Level}'(x) < \textsc{Level}(x)$. Choose v to be one among them with minimum $\textsc{Level}'(v)$. By the choice of v, and Definition 4.3.4(3), $\textsc{Level}'(v) > 0$ and $v \in \mathbb{A}$ after the Push operation. Thus, by Proposition 4.3.3(4), there exists a node u with $\textsc{Level}'(u) = \textsc{Level}'(v)-1$, so that after Push,

- Case 1: v is reachable from $u \in \mathbb{A}$, or
- Case 2: there exists another node $v' \in \mathbb{A}$ reachable from $u \in \mathbb{A}$ with $\textsc{Level}'(v') = \textsc{Level}'(v)$ in the same system as v, and v is critical.

Notice that by the choice of v, in both cases, $\textsc{Level}'(u) \geq \textsc{Level}(u)$, and $u \in \mathbb{A}$ also before the Push operation. Let p be the pebble by which u reaches v (Case 1), or v' (Case 2), after Push. Before the Push operation, p was at some node $u' \in \mathbb{A}$ (u' may be u, or p is the pebble pushed: from u' to u).

By Proposition 4.3.3(2), in Case 1, $\textsc{Level}(v) \leq \textsc{Level}(u') + 1$ (as v is reachable from u' via p before Push), and $\textsc{Level}(v') \leq \textsc{Level}(u')+1$ in Case 2. Furthermore, if in Case 2 v was already critical before push, then $\textsc{Level}(v) \leq \textsc{Level}(v')$ by Proposition 4.3.3(3) (note that $v' \in \mathbb{A}$ as it is reachable from $u' \in \mathbb{A}$). Hence, in both cases we would have

$$\textsc{Level}(v) \leq \textsc{Level}(u') + 1 \leq \textsc{Level}(u) + 1 \leq \textsc{Level}'(u) + 1 = \textsc{Level}'(v),$$

a contradiction. Note that the second inequality holds no matter $u = u'$ or not.

Finally consider Case 2 where v was not critical before the Push operation. Then a pebble $p' \neq p$ is pushed into v in the operation. Note that in this situation, v's system is a tree and contains no critical nodes before Push (by Definition 4.3.4(3)); in particular v' is not critical. Furthermore, the presence of p in u implies that $\textsc{Level}(v') \leq \textsc{Level}(u) + 1$ by Proposition 4.3.3(2), and that $v' \in \mathbb{A}$ by Proposition 4.3.3(1). As v' is not critical, $\textsc{Level}(v') > 0$, and by Proposition 4.3.3(4) there exists a node u'' with $\textsc{Level}(u'') = \textsc{Level}(v') - 1$ so that u'' can reach v' by a pebble p'' (u'' may be u and p'' may be p). As

$$\textsc{Level}(v') \leq \textsc{Level}(u) + 1 \leq \textsc{Level}'(u) + 1 = \textsc{Level}'(v) < \textsc{Level}(v),$$

the Push operation should have pushed p'' into v' instead of p' into v (see Definition 4.3.4(4)), since u'' and v' satisfy all the first three conditions of Definition 4.3.4. □

By Lemma 4.3.7 and the fact that a pebble is pushed to a node with higher level, the potential Φ strictly decreases after each Push operation, implying that Algorithm 1 finishes in polynomial time.

Approximation Ratio: When $t < 2W$, we apply Algorithm 1. In the case of $t \geq 2W$, we apply the algorithm of Gairing et al. [39], which either correctly reports that OPT $\geq t+1$, or returns an assignment with makespan at most $t+W-1 < 1.5t$.

Suppose that t is the smallest number for which an assignment is returned. Then $\mathrm{OPT} \geq t$, and our approximation ratio is bounded by $\frac{1.5t}{\mathrm{OPT}} \leq 1.5$. We summarize our result in a theorem.

Theorem 4.3.8. *With arbitrary dedicated loads on the machines, jobs of weight W that can be assigned to two machines, and jobs of weight w that can be assigned to any number of machines, we can find a* 1.5 *approximate solution in polynomial time.*

In the following, we show that a slight modification of our algorithm yields an improved approximation ratio of $1 + \frac{\lfloor \frac{W}{2} \rfloor}{W}$ if $W \geq 2w$.

Improving the Ratio. Suppose that $W \geq 2w$.

As before, we first assume that $t < 2W$, and discuss the case $t \geq 2W$ at the end of the section. We modify our previous algorithm as follows:

Definition 4.3.9. *A node v is*

- uncritical*, if $dl(v) + pl(v) \leq t + \lfloor \frac{W}{2} \rfloor - W - w$;*
- critical*, if $dl(v) + pl(v) > t + \lfloor \frac{W}{2} \rfloor - W$;*
- hypercritical*, if $dl(v) + pl(v) > t + \lfloor \frac{W}{2} \rfloor$.*

> **Modified Algorithm 1**: As long as there is a bad system, apply EXPLORE1 and PUSH operation repeatedly. When there is no bad system left, return a solution with makespan at most $t + \lfloor \frac{W}{2} \rfloor$. If at some point, PUSH is no longer possible, declare that $\mathrm{OPT} \geq t + 1$.

The proof of Lemma 4.3.7 remains the same, and to establish Lemma 4.3.5 we just need to re-do the proof of Claim 4.3.6.

New Proof of Claim 4.3.6: By the same reasoning as before,

- none of the nodes in $\mathbb{A}(S)$ is uncritical;
- if S is a tree and $\mathbb{A}(S) \neq \emptyset$, at least one node $v \in \mathbb{A}(S)$ is critical; furthermore, if $|\mathbb{A}(S)| = 1$, this node v satisfies $dl(v) + pl(v) > t + \lfloor \frac{W}{2} \rfloor - w$;
- if S is an isolated node $v \in \mathbb{A}$, then $dl(v) + pl(v) > t + \lfloor \frac{W}{2} \rfloor - w$.

We now re-do the case analysis.

1. Suppose that S is a good system and $\mathbb{A}(S) \neq \emptyset$. Then either S is a tree and $\mathbb{A}(S)$ contains exactly one critical (but not hypercritical) node, or S is

an isolated node, or S is a cycle and has no critical node. In the first case, if $|\mathbb{A}(S)| \geq 2$, the LHS of (4.2) is at least

$$(t + \lfloor \frac{W}{2} \rfloor - W + 1) + (|\mathbb{A}(S)| - 1)(t + \lfloor \frac{W}{2} \rfloor - W - w + 1) + (|\mathbb{A}(S)| - 1)W =$$
$$|\mathbb{A}(S)|t - W + |\mathbb{A}(S)|(\lfloor \frac{W}{2} \rfloor + 1) - (|\mathbb{A}(S)| - 1)w >$$
$$|\mathbb{A}(S)|t + \frac{(|\mathbb{A}(S)| - 2)W}{2} - (|\mathbb{A}(S)| - 1)w \geq |\mathbb{A}(S)|t - w,$$

where the first inequality holds because $\lfloor \frac{W}{2} \rfloor + 1 > \frac{W}{2}$ and the last inequality holds because $|\mathbb{A}(S)| \geq 2$ and $W \geq 2w$. If, on the other hand, $|\mathbb{A}(S)| = 1$, then the LHS of (4.2) is strictly more than

$$t + \lfloor \frac{W}{2} \rfloor - w \geq t = |\mathbb{A}(S)|t,$$

and the same also holds for the case when S is an isolated node. Finally, in the third case, the LHS of (4.2) is at least

$$|\mathbb{A}(S)|(t + \lfloor \frac{W}{2} \rfloor - W - w + 1) + |\mathbb{A}(S)|W > |\mathbb{A}(S)|t.$$

2. Suppose that $\mathbb{A}(S)$ contains at least two critical nodes, or that S is a cycle and $\mathbb{A}(S)$ has at least one critical node. In both cases, S is a bad system. Furthermore, the LHS of (4.1) can be lower-bounded by the same calculation as in the previous case with an extra term of w.

3. Suppose that $\mathbb{A}(S)$ contains a hypercritical node. Then the system S is bad, and the LHS of (4.1) is at least

$$(t + \lfloor \frac{W}{2} \rfloor + 1) + (|\mathbb{A}(S)| - 1)(t + \lfloor \frac{W}{2} \rfloor - W - w + 1) + (|\mathbb{A}(S)| - 1)W =$$
$$|\mathbb{A}(S)|(t + \lfloor \frac{W}{2} \rfloor + 1) - (|\mathbb{A}(S)| - 1)w > |\mathbb{A}(S)|t,$$

where the last inequality holds because $W \geq 2w$. ☐

Approximation Ratio: When $t \geq 2W$, we can again use the Gairing et al's algorithm [39], which either correctly reports that $\text{OPT} \geq t + 1$, or returns an assignment with makespan at most $t + W - 1$.

Suppose that t is the smallest number for which an assignment is returned (then $\text{OPT} \geq t$). Then the approximation ratio is

$$\frac{t + \lfloor \frac{W}{2} \rfloor}{\text{OPT}}, \text{if } t < 2W; \qquad \frac{t + W - 1}{\text{OPT}}, \text{if } t \geq 2W.$$

The former is bounded by $1 + \frac{\lfloor \frac{W}{2} \rfloor}{W}$, since $\text{OPT} \geq W$; the latter is bounded by $1 + \frac{W-1}{2W} \leq 1 + \frac{\lfloor \frac{W}{2} \rfloor}{W}$, since $\text{OPT} \geq t \geq 2W$. We can thus conclude:

Theorem 4.3.10. *Suppose that $W \geq 2w$. With arbitrary dedicated loads on the machines, jobs of weight W that can be assigned to two machines, and jobs of weight w that can be assigned to any number of machines, we can find a $1+\frac{\lfloor \frac{W}{2} \rfloor}{W}$ approximate solution in polynomial time.*

4.4. The General Case

In this section, we describe the core algorithm for the case of arbitrary job weights. This algorithm inherits some basic ideas from the previous section, but has several significantly new ingredients—mainly due to the fact that the rocks now have different weights. Before formally presenting the algorithm, let us build up intuition by looking at some examples.

For simplicity, we rescale the numbers and assume that $t = W = 1$ and $\beta = 0.7$. We aim for an assignment with makespan of at most $5/3+0.7/3 = 1.9$ or decide that OPT > 1. Consider the example in Figure 4.2. Note that there are $2k+1$ (for some large k) nodes (the pattern of the last two nodes repeats). Due to node 1 (which can be regarded as the analog of a critical node in the previous section), all edges are to be directed toward the right if we shoot for the makespan of 1.9. Suppose that there is an isolated node with the pebble load of $2 + \epsilon$ (this node can be regarded as a bad system by itself) and it has a pebble of weight 0.7 that can be assigned to node 3, 5, 7 and so on up to $2k + 1$. Clearly, we do not want to push the pebble into any of them, as it would cause the makespan to be larger than 1.9 by whatever orientation. Rather, we should activate node 1 and send its pebbles away with the aim of relieving the "congestion" in the current system (later we will see that this is activation rule 1). In this example, all odd-numbered nodes are activated, and the entire set of nodes (including even-numbered nodes) form a *conflict set* (which will be defined formally later). Roughly speaking, the conflict sets contain activated nodes and the nodes that can be reached by "backtracking" the directed edges from them. These conflict sets embody the "congestion" in the systems.

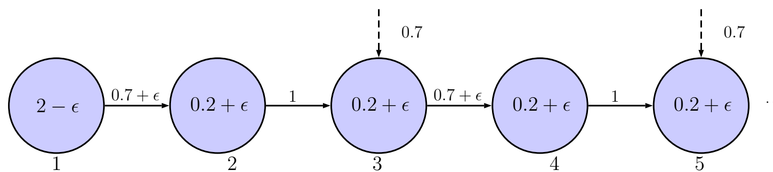


Figure 4.2.: There are $2k+1$ nodes (the rest is repeating the same pattern). Numbers inside the shaded circles (nodes) are their pebble load.

Recall that in the previous section, if the PUSH operation was no longer possible, we argued that the total load is too much (see the proof of Lemma 4.3.5) for the activated nodes *system by system*. Analogously, in this example, we need to argue that in all feasible orientations, the activated set of nodes (totally $k + 1$ of them)

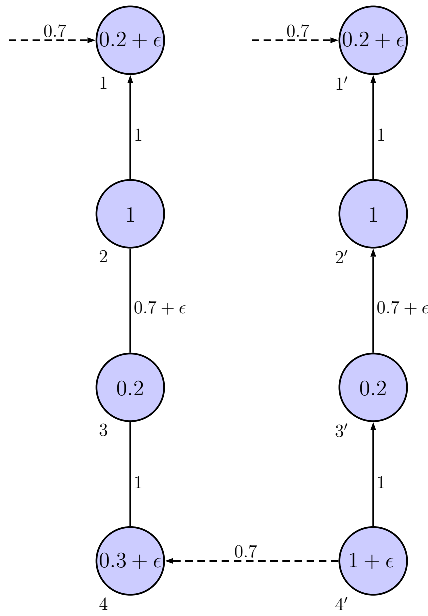


Figure 4.3.: A naive Push will oscillate the pebble between nodes 4 and $4'$.

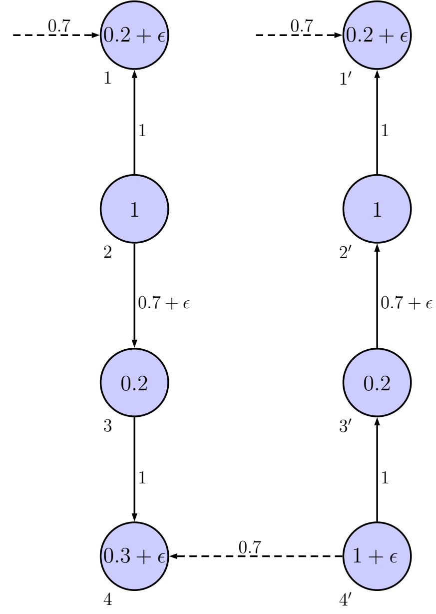


Figure 4.4.: A fake orientation from node 2 to 3 causes node 4 to have an incoming edge, thus informing node $4'$ not to push the pebble.

in this conflict set cannot handle the total load. However, if all edges are directed toward the left, their total load is only $(0.2+\epsilon)k + (2-\epsilon) + (0.7+\epsilon)k = 2 + 0.9k + \epsilon(2k-1)$, which is less than what they can handle (which is $k+1$) when k is large. As a result, we are unable to arrive at a contradiction.

To overcome this issue, we introduce another activation rule to strengthen our contradiction argument. If all edges are directed to the left, *on the average*, each activated node has a total load of about $0.2 + 0.7$. However, each inactivated node has, *on the average*, a total load of about $0.2 + 1$. This motivates our activation rule 2 : if an activated node is connected by a "relatively light" edge to some other node in the conflict set, the latter should be activated as well. The intuition behind is that the two nodes *together* will receive a relatively heavy load. We remark that it is easy to modify this example to show that if we do not apply activation rule 2, then we cannot hope for a $2-\delta$ approximation for any small $\delta > 0$. [3]

Next consider the example in Figure 4.3. Here nodes 2, $2'$, and $4'$ can be regarded

[3] Looking at this particular example, one is tempted to use the idea of activating all nodes in the conflict set. However, such an activation rule will not work. Consider the following example: There are $k+2$ nodes forming a path, and the $k+1$ edges connecting them all have weight $0.95+\epsilon$. The first node has a pebble load of 1 and thus "forces" an orientation of the entire path (for a makespan of at most 1.9). The next k nodes have a pebble load of 0, and the last node has a pebble load of 0.25 and is reachable from a bad system via a pebble of weight 0.7. The conflict set is the entire path, and activating all nodes leads to a total load of $(k+1)\cdot(0.95+\epsilon)+1+0.25$, which is less than $k+2$ for large k.

as the critical nodes, and $\{1,2\}$, $\{1',2',3',4'\}$ are the two conflict sets. Both nodes 1 and $1'$ can be reached by an isolated node with heavy load (the bad system) with a pebble of weight 0.7. Suppose further that node $4'$ can reach node 4 by another pebble of weight 0.7. It is easy to see that a naive PUSH definition will simply "oscillate" the pebble between nodes 4 and $4'$, causing the algorithm to cycle.

Intuitively, it is not right to push the pebble from $4'$ into 4, as it causes the conflict set in the left system to become bigger. Our principle of pushing a pebble should be to relieve the congestion in one system, while not worsening the congestion in another. To cope with this problematic case, we use *fake orientations*, i.e., we direct edges away from a conflict set, as shown in Figure 4.4. Node 2 directs the edge toward node 3, which in turn causes the next edge to be directed toward node 4. With the new incoming edge, node 4 now has a total load of $1+0.3+\epsilon$ to handle, and the pebble thus will not be pushed from node $4'$ to node 4.

4.4.1. Formal description of the algorithm

We inherit some terminology from the previous section. We say that v is *reachable* from u if a pebble in u can be assigned to v, and that v is reachable from $\mathbb{A}$ if v is reachable from any node $u \in \mathbb{A}$. Each tree, cycle, isolated node in $G_{\mathbb{R}}$ is a system. Note that there is exactly one edge between two adjacent nodes in $G_{\mathbb{R}}$ (see Proposition 4.2.1). For ease of presentation, we use the short hand vu to refer to the edge $\{v,u\}$ in $G_{\mathbb{R}}$ and w_{vu} is its weight.

The orientation of the edges in $G_{\mathbb{R}}$ will be decided dynamically. If uv is directed toward v, we call v a *father* of u, and u a *child* of v (notice that a node can have several fathers and children). We write $rl(v)$ to denote total weight of the rocks that are (currently) oriented towards v, and $pl(v)$ still denotes the total weight of the pebbles at v. An edge that is currently un-oriented is *neutral*. In the beginning, all edges in $G_{\mathbb{R}}$ are neutral.

A set $\mathbb{C}$ of nodes, called the *conflict set*, will be collected in the course of the algorithm. Let $\mathcal{D}(v) := \{u \in \mathbb{C} : u \text{ is child of } v\}$ and $\mathcal{F}(v) := \{u \in \mathbb{C} : u \text{ is father of } v\}$ for any $v \in \mathbb{C}$. A node $v \in \mathbb{C}$ is a *leaf* if $\mathcal{D}(v) = \emptyset$, and a *root* if $\mathcal{F}(v) = \emptyset$. Furthermore, a node v is *overloaded* if $dl(v)+pl(v)+rl(v) > (5/3+\beta/3)t$, and a node $v \in \mathbb{C}$ is *critical* if there exists $u \in \mathcal{F}(v)$ such that $dl(v)+pl(v)+w_{vu} > (5/3+\beta/3)t$. In other words, a node in the conflict set is critical if it has enough load by itself (without considering incoming rocks) to "force" an incident edge to be directed toward a father in the conflict set.

Initially, the pebbles are arbitrarily assigned to the nodes. The orientation of a subset of the edges in $G_{\mathbb{R}}$ is determined by the procedure FORCED ORIENTATIONS in Figure 4.5.

Intuitively, the procedure first finds a "source node" v, whose dedicated, pebble, and rock load is so high that it "forces" an incident edge vu to be oriented away from v. The orientation of this edge then propagates through the graph, i.e. edge-orientations induced by the direction of vu are established. Then the next "source" is found, and so on. To simplify our proofs, we assume that ties are broken according

FORCED ORIENTATIONS

While $\exists$ neutral edge vu in $G_{\mathbb{R}}$, s.t. $dl(v) + pl(v) + rl(v) + w_{vu} > (5/3 + \beta/3)t$:

 Direct vu towards u; MARKED := $\{u\}$.

 While $\exists$ neutral edge $v'u'$ in $G_{\mathbb{R}}$, s.t. $dl(v') + pl(v') + rl(v') + w_{v'u'} > (5/3 + \beta/3)t$ and $v' \in$ MARKED:

  Direct $v'u'$ towards u'; MARKED := MARKED $\cup \{u'\}$.

Figure 4.5.: The procedure FORCED ORIENTATIONS.

to a fixed total order if several pairs (v, u) satisfy the conditions of the while-loops.

The following lemma describes a basic property of the procedure FORCED ORIENTATIONS, that will be used in the subsequent discussion.

Lemma 4.4.1. *Suppose that a node v becomes overloaded during* FORCED ORIENTATIONS. *Then there exists a path $u_0u_1 \dots u_kv$ of neutral edges, such that $dl(u_0) + pl(u_0) + rl(u_0) + w_{u_0u_1} > (5/3 + \beta/3)t$ before the procedure, that becomes directed from u_0 towards v during the procedure (note that u_0 could be v). Furthermore, other than u_kv, no edge becomes directed toward v in the procedure.*

Proof. We start with a simple observation. Let ab be the first edge directed in some iteration of the procedure's outer while-loop; suppose from a to b. It is easy to see that up to this moment, no edge has been directed toward a in course of the procedure. Furthermore, if another edge $a'b'$ is directed in the same iteration of the outer while-loop, then there exists a path of neutral edges, starting with ab and ending with $a'b'$, that becomes directed during this iteration. This proves the first part of the lemma.

Now suppose that some node v becomes overloaded and has more than one edge directed towards it during the procedure. Let vx and vy be the last two edges directed toward v, and note that both, vx and vy, become directed in the same iteration of the outer while-loop (because as soon as one of the two is directed toward v, the other edge satisfies the conditions of the inner while-loop). Hence, there are two different paths directed towards v (with final edges vx and vy, respectively), both of which start with the first edge that becomes directed in this iteration of the outer while-loop. This is not possible, since every system is a tree or a cycle, a contradiction. □

Clearly, if after the procedure FORCED ORIENTATIONS a node v still has a neutral incident edge vu, then $dl(v) + pl(v) + rl(v) + w_{vu} \leq (5/3 + \beta/3)t$. Now suppose that after the procedure, none of the nodes is overloaded. Then orienting the neutral edges in each system in such a way that every node has at most one more incoming edge gives us a solution with makespan at most $(5/3 + \beta/3)t$. So assume the procedure ends with a non-empty set of overloaded nodes. We then apply the procedure EXPLORE2 in Figure 4.6.

EXPLORE2

Initialize $\mathbb{A} := \emptyset$; $\mathbb{C} := \emptyset$; $i := 0$. **Call** FORCED ORIENTATIONS.
Repeat:
 If $i = 0$: $\mathbb{A}_i := \{v | v \text{ is overloaded}\}$.
 Else $\mathbb{A}_i := \{v | v \notin \mathbb{A}, v \text{ is reachable from } \mathbb{A}_{i-1}\}$.
 If $\mathbb{A}_i = \emptyset$: **stop**.
 $\mathbb{C}_i := \mathbb{A}_i$; $\mathbb{A} := \mathbb{A} \cup \mathbb{A}_i$; $\mathbb{C} := \mathbb{C} \cup \mathbb{C}_i$.

 (*Conflict set construction*)
 While $\exists v \notin \mathbb{C}$ with a father $u \in \mathbb{C}$ **or** $\exists$ neutral vu with $v \in \mathbb{C}$ **do:**
  While $\exists v \notin \mathbb{C}$ with a father $u \in \mathbb{C}$:
   $\mathbb{C}_i := \mathbb{C}_i \cup \{v\}$; $\mathbb{C} := \mathbb{C} \cup \mathbb{C}_i$.
  If $\exists$ neutral vu with $v \in \mathbb{C}$:
   Direct vu towards u; **Call** FORCED ORIENTATIONS.

 (*Activation of nodes*)
 While $\exists v \in \mathbb{C} \setminus \mathbb{A}$ satisfying one of the following conditions:
  Rule 1: $\exists u \in \mathcal{F}(v)$, such that $dl(v) + pl(v) + w_{vu} > (5/3 + \beta/3)t$
  Rule 2: $\exists u \in \mathbb{A} \cap (\mathcal{D}(v) \cup \mathcal{F}(v))$, such that $w_{vu} < (2/3 + \beta/3)t$
 Do: $\mathbb{A}_i := \mathbb{A}_i \cup \{v\}$; $\mathbb{A} := \mathbb{A} \cup \mathbb{A}_i$.

 $i := i + 1$.

Figure 4.6.: The procedure EXPLORE2.

Let us elaborate the procedure. In each round, we perform the following three tasks.

1. Add those nodes reachable from the nodes in $\mathbb{A}_{i-1}$ into $\mathbb{A}_i$ in case of $i > 1$; or the overloaded nodes into $\mathbb{A}_i$ in case of $i = 0$. These nodes will be referred to as Type A nodes.

2. In the sub-procedure *Conflict set construction*, nodes not in the conflict set and having a directed path to those Type A nodes in $\mathbb{A}_i$ are continuously added into the conflict set $\mathbb{C}_i$. Furthermore, the earlier mentioned *fake orientations* are applied: each node $v \in \mathbb{C}_i$, if having an incident neutral edge vu, direct it toward u and call the procedure FORCED ORIENTATIONS. It may happen that in this process, two disjoint nodes in $\mathbb{C}_i$ are now connected by a directed path P, then all nodes in P along with all nodes having a path leading to P are added into $\mathbb{C}_i$ (observe that all these nodes have a directed path to some Type A node in $\mathbb{A}_i$). We note that the order of fake orientations does not materially affect the outcome of the algorithm (see Lemma 4.4.21).

3. In the next sub-procedure *Activation of nodes*, we use two rules to activate extra nodes in $\mathbb{C} \backslash \mathbb{A}$. Rule 1 activates the critical nodes; Rule 2 activates those

nodes whose father or child are already activated and they are connected by an edge of weight less than $(2/3 + \beta/3)t$. We will refer to the former as Type B nodes and the latter as Type C nodes.

Observe that except in the initial call of FORCED ORIENTATIONS, no node ever becomes overloaded in EXPLORE2 (by Lemma 4.4.1 and the fact that every system is a tree or a cycle). Let us define $\text{LEVEL}(v) = i$ if $v \in \mathbb{A}_i$. In case $v \notin \mathbb{A}$, let $\text{LEVEL}(v) = \infty$. The next proposition summarizes some important properties of the procedure EXPLORE2.

Proposition 4.4.2. *After the procedure* EXPLORE2*, the following holds.*

1. *All nodes reachable from $\mathbb{A}$ are in $\mathbb{A}$.*

2. *Suppose that $v \in \mathbb{A}$ is reachable from $u \in \mathbb{A}$. Then* $\text{LEVEL}(v) \leq \text{LEVEL}(u) + 1$.

Furthermore, at the end of each round i, the following holds.

3. *Every node v that can follow a directed path to a node in $\mathbb{C} := \cup_{\tau=0}^{i} \mathbb{C}_\tau$ is in $\mathbb{C}$. Furthermore, if a node $v \in \mathbb{C}$ has an incident edge vu with $u \notin \mathbb{C}$, then vu is directed toward u.*

4. *Each node $v \in \mathbb{A}_i$ is one of the following three types.*

   a) **Type A***: there exists another node $u \in \mathbb{A}_{i-1}$ so that v is reachable from u, or v is overloaded and is part of $\mathbb{A}_0$.*

   b) **Type B***: v is activated via Rule 1 (hence v is critical)[4], and there exists a directed path from v to $u \in A_i$ of Type A.*

   c) **Type C***: v is activated via Rule 2, and there exists an adjacent node $u \in \cup_{\tau=0}^{i} \mathbb{A}_\tau$ so that $w_{vu} < (2/3 + \beta/3)t$ and $u \in \mathcal{D}(v) \cup \mathcal{F}(v)$.*

After the procedure EXPLORE2, we apply the PUSH operation (if possible), defined as follows.

Definition 4.4.3. PUSH *operation: push a pebble from u^* to v^* if the following conditions hold (if there are multiple candidates, pick any).*

1. *The pebble is at u^* and it can be assigned to v^*.*

2. $\text{LEVEL}(v^*) = \text{LEVEL}(u^*) + 1$.

3. $dl(v^*) + pl(v^*) + rl(v^*) \leq (5/3 - 2/3 \cdot \beta)t$.

4. $\mathcal{D}(v^*) = \emptyset$*, or* $dl(v^*) + pl(v^*) + w_{v^*u} \leq (5/3 - 2/3 \cdot \beta)t$ *for all* $u \in \mathcal{F}(v)$.

[4] For simplicity, if a node can be activated by both Rule 1 and Rule 2, we assume it is activated by Rule 1.

Definition 4.4.3(3) is meant to make sure that v^* does not become overloaded after receiving a new pebble (whose weight can be as heavy as βt). Definition 4.4.3(4) says either v^* is a leaf, or adding a pebble with weight as heavy as βt does not cause v^* to become critical.

> **Algorithm 2**: Apply EXPLORE2. If it ends with $\mathbb{A}_0 = \emptyset$, return a solution with makespan at most $(5/3 + \beta/3)t$. Otherwise, apply PUSH. If PUSH is impossible, declare that OPT $\geq t + 1$. Un-orient all edges in $G_{\mathbb{R}}$ and repeat this process.

Lemma 4.4.4. *When there is at least one overloaded node and the* PUSH *operation is no longer possible,* OPT $\geq t + 1$.

Lemma 4.4.5. *For each node $v \in V$, let* $\text{LEVEL}(v)$ *and* $\text{LEVEL}'(v)$ *denote the levels before and after a* PUSH *operation, respectively. Then* $\text{LEVEL}'(v) \geq \text{LEVEL}(v)$.

The preceding two lemmas are proven in sections 4.4.2 and 4.4.3, respectively. We again use the potential function

$$\Phi = \sum_{v \in \mathbb{A}} (|V| - \text{LEVEL}(v)) \cdot (\text{number of pebbles at } v)$$

to argue the polynomial running time of Algorithm 2. Trivially, $0 \leq \Phi \leq |V| \cdot |\mathbb{P}|$. Furthermore, by Lemma 4.4.5 and the fact that a pebble is pushed to a node with higher level, the potential Φ strictly decreases after each PUSH operation. This implies that Algorithm 2 finishes in polynomial time.

We can therefore conclude:

Theorem 4.4.6. *Let $\beta \in [4/7, 1)$. With arbitrary dedicated loads on the machines, if jobs of weight greater than βW can be assigned to only two machines, and jobs of weight at most βW can be assigned to any number of machines, we can find a $5/3 + \beta/3$ approximate solution in polynomial time.*

4.4.2. Proof of Lemma 4.4.4

Our goal is to show that in any feasible solution, the activated nodes $\mathbb{A}$ must handle a total load of more than $|\mathbb{A}|t$, which implies that OPT $\geq t + 1$. For the proof, we focus on a single component K of $G_{\mathbb{R}}[\mathbb{C}]$, the subgraph of $G_{\mathbb{R}}$ induced by the conflict set $\mathbb{C}$, and a fixed orientation $\psi \in \Psi$. Let $\psi(v)$ denote the total weight of the rocks assigned to any $v \in \mathbb{A}$ by ψ (note that $0 \leq \psi(v) \leq t$), and let $\mathbb{A}(K)$ denote the set of activated nodes in K. We will show that

$$\sum_{v \in \mathbb{A}(K)} pl(v) + dl(v) + \psi(v) > |\mathbb{A}(K)|t \tag{4.3}$$

if $\mathbb{A}(K) \neq \emptyset$. The lemma then follows by summing over all components of $G_{\mathbb{R}}[\mathbb{C}]$, and noting that the pebbles on the nodes in $\mathbb{A}$ can only be assigned to the nodes in $\mathbb{A}$ (Proposition 4.4.2(1)).

If K consists only of a single activated node v, then (4.3) clearly holds, as $pl(v) + dl(v) > (5/3 - 2/3 \cdot \beta)t \geq t$ (since v is a Type A node and PUSH is no longer possible). In the following, we will assume that $\mathcal{F}(v) \cup \mathcal{D}(v) \neq \emptyset$ for all $v \in \mathbb{A}(K)$.

Definition 4.4.7. *For every non-leaf $v \in \mathbb{A}(K)$, fix some node $d(v) \in \mathcal{D}(v)$, such that $w_{vd(v)} = \max_{u \in \mathcal{D}(v)} w_{vu}$.*

Definition 4.4.8. *For every non-root $v \in \mathbb{A}(K)$, fix some node $f(v) \in \mathcal{F}(v)$, such that $w_{vf(v)} = \max_{u \in \mathcal{F}(v)} w_{vu}$.*

Definition 4.4.9. *For every node $v \in \mathbb{A}(K)$ that is neither a root nor a leaf, fix some node $n(v) \in \mathcal{D}(v) \cup \mathcal{F}(v)$, such that $w_{vn(v)} = \max_{u \in \mathcal{D}(v) \cup \mathcal{F}(v)} w_{vu}$.*

Definition 4.4.10. *For every node $v \in \mathbb{A}(K)$ that was activated using Rule 2 in the final execution of* EXPLORE2*, fix some node $a(v) \in \mathbb{A}(K) \cap (\mathcal{D}(v) \cup \mathcal{F}(v))$ with $w_{va(v)} < (2/3 + \beta/3)t$, such that $a(v)$ has been activated before v.*

We classify the nodes $v \in \mathbb{A}(K)$ that are neither a root nor a leaf, into the following three types.

Type 1: $|\mathcal{D}(v)| > 1$.

Type 2: $|\mathcal{D}(v)| = 1$ and v was activated via Rule 2 (i.e., as a Type C node).

Type 3: $|\mathcal{D}(v)| = 1$ and v was not activated via Rule 2 (i.e. as a Type A or Type B node).

In the following, we summarize the inequalities that we use for the different types of nodes, in order to prove (4.3). We refer to them as the *load-inequalities.*

Claim 4.4.11. *For every leaf $v \in \mathbb{A}(K)$, $pl(v) + dl(v) > (5/3 + \beta/3)t - w_{vf(v)}$.*

Proof. If $v \in \mathbb{A}_i$ is activated as a Type A node, then it is either overloaded or is reachable from a node $u \in \mathbb{A}_{i-1}$. In both cases, since PUSH is no longer possible, $pl(v)+dl(v)+rl(v) > (5/3-2/3\cdot\beta)t$. The claim follows as $rl(v) = 0$ and $w_{vf(v)} > \beta t$. If v is not activated as a Type A node, then v first becomes part of $\mathbb{C}$ and then becomes activated via Rule 1 or Rule 2. In this case, at the moment v becomes part of $\mathbb{C}$, it must have a father $u \in \mathbb{C}$. The edge vu becomes oriented towards u only when FORCED ORIENTATIONS is called and $dl(v)+pl(v)+rl(v)+w_{vu} > (5/3+\beta/3)t$. The claim follows again as $rl(v) = 0$ and $w_{vf(v)} \geq w_{vu}$. □

Claim 4.4.12. *For every root $v \in \mathbb{A}(K)$ with $|\mathcal{D}(v)| = 1$, $pl(v) + dl(v) > (5/3 - 2/3 \cdot \beta)t - w_{vd(v)}$.*

Proof. As $v \in \mathbb{A}_i$ has no father in $\mathbb{C}$, it must either be overloaded or reachable from an activated node $u \in \mathbb{A}_{i-1}$. In both cases, $pl(v) + dl(v) + rl(v) > (5/3 - 2/3 \cdot \beta)t$, since the PUSH operation is no longer possible. The claim follows as $|\mathcal{D}(v)| = 1$ implies $w_{vd(v)} \geq rl(v)$. □

Claim 4.4.13. *For every root $v \in \mathbb{A}(K)$ with $|\mathcal{D}(v)| > 1$, $pl(v) + dl(v) \geq 0$.*

Proof. Trivially true. □

Claim 4.4.14. *For every Type 1 node $v \in \mathbb{A}(K)$, $pl(v) + dl(v) \geq 0$.*

Proof. Trivially true. □

Claim 4.4.15. *For every Type 2 node $v \in \mathbb{A}(K)$, $pl(v) + dl(v) > (5/3 + \beta/3)t - w_{vf(v)} - w_{vd(v)}$.*

Proof. As v is activated using Rule 2, it first becomes part of $\mathbb{C}$ without being activated. For this to happen, it must have a father $u \in \mathbb{C}$. The edge vu becomes oriented towards u only when Forced Orientations is called and $dl(v) + pl(v) + rl(v) + w_{vu} > (5/3 + \beta/3)t$. The claim follows as $w_{vd(v)} \geq rl(v)$ (since $|\mathcal{D}(v)| = 1$) and $w_{vf(v)} \geq w_{vu}$. □

Claim 4.4.16. *For every Type 3 node $v \in \mathbb{A}(K)$, $pl(v) + dl(v) > (5/3 - 2/3 \cdot \beta)t - w_{vn(v)}$.*

Proof. If v is overloaded, the claim directly follows from the fact that $w_{vn(v)} \geq rl(v)$. Furthermore, if $v \in A_i$ is reachable from an activated node $u \in A_{i-1}$, then the claim follows from the definition of $n(v)$ and the fact that either the third or the fourth condition of push must be violated. The only other possibility for v to be activated is via Rule 1, which together with the definition of $n(v)$ implies our claim. □

To prove (4.3), we look at each node $v \in \mathbb{A}(K)$ separately and calculate how much it contributes to the balance under some simplifying assumptions. In the end, we will see that the nodes in $\mathbb{A}(K)$ have enough load to compensate for the assumptions we made.

Let $E_{\mathbb{A}(K)}$ denote the edges of K that are incident with the nodes $\mathbb{A}(K)$, i.e. $E_{\mathbb{A}(K)} := \{vu \in \mathbb{R} : u \in \mathbb{A}(K), v \in \mathcal{D}(u) \cup \mathcal{F}(u)\}$. We say that an edge $vu \in E_{\mathbb{A}(K)}$ is *covered* if w_{vu} appears on the right-hand side of u's and/or v's load-inequality. For example, if v is a leaf, then $vf(v)$ is covered. Every edge in $E_{\mathbb{A}(K)}$ that is not covered is called *uncovered.* Finally, we say that an edge $vu \in E_{\mathbb{A}(K)}$ is *doubly covered* if w_{vu} appears on the right-hand side of both u's and v's load-inequality.

We distinguish two cases.

Case 1: K is a tree.

Claim 4.4.17. *K contains $1 + \sum_{v \in K : \mathcal{F}(v) \neq \emptyset}(|\mathcal{F}(v)| - 1)$ many roots, and $1 + \sum_{v \in K : \mathcal{D}(v) \neq \emptyset}(|\mathcal{D}(v)| - 1)$ many leaves. Furthermore, every root and leaf in K is activated.*

Proof. The first part simply follows from the degree sum formula for directed graphs and the fact that K is a tree. For the second part, observe that any node $v \in \mathbb{C}$ that is not activated as Type A node, must have had a father $u \in \mathbb{C}$ already before

it got added into $\mathbb{C}$ itself. This proves that every root in K is activated (as a Type A node).

If a leaf $v \in \mathbb{C}$ is not activated as Type A node, then its incident edge vu with $u \in \mathbb{C}$ is oriented toward u only when FORCED ORIENTATIONS is called and $dl(v) + pl(v) + rl(v) + w_{vu} > (5/3 + \beta/3)t$. As $v \in \mathbb{C}$ ends up a leaf, $rl(v) = 0$, and Rule 1 would have applied to v. So every leaf in K is activated. □

In our calculations, we will assume that every covered edge $vu \in E_{\mathbb{A}(K)}$ has weight $w_{vu} = t$, and that $\psi(v) = t$ for all $v \in \mathbb{A}(K)$. With these assumptions, we will show that

$$\begin{aligned}\sum_{v\in\mathbb{A}(K)} pl(v) + dl(v) + \psi(v) > |\mathbb{A}(K)|t \\ &- |\{\text{doubly covered } vu \in E_{\mathbb{A}(K)} : w_{vu} < (2/3 + \beta/3)t\}| \cdot (1/3 - \beta/3)t \\ &+ |\{\text{uncovered } vu \in E_{\mathbb{A}(K)}\}| \cdot (t - w_{vu}) \\ &+ t.\end{aligned} \tag{4.4}$$

Let us consider the error caused by these two assumptions when we lower-bound the term $\sum_{v\in\mathbb{A}(K)} pl(v) + dl(v) + \psi(v)$, and in doing so, we will show why (4.4) implies (4.3).

Consider an edge $vu \in E_{\mathbb{A}(K)}$ that ψ assigns to a node in $\mathbb{A}(K)$, say v. Consider three possibilities.

- If vu is covered, then w_{vu} appears on the LHS of (4.3) as a negative term after we plug in the load-inequalities, and the two terms $\psi(v)$ and w_{vu} cancel each other. Hence, in this case, we make no error by assuming both terms to be equal to t.
- If vu is doubly covered and $w_{vu} < (2/3+\beta/3)t$, our assumptions underestimate the load $\sum_{v\in\mathbb{A}(K)} pl(v) + dl(v) + \psi(v)$ by more than $(1/3 - \beta/3)t$.
- If vu is uncovered, then we overestimate $\psi(v)$ by at most $t - w_{vu}$.

Finally, we note that ψ must assign an edge from $E_{\mathbb{A}(K)}$ to every node in $\mathbb{A}(K)$ except for possibly one. For this special node v^* that does not receive an edge from $E_{\mathbb{A}(K)}$ under ψ, we overestimate $\psi(v^*)$ by at most t. In conclusion, when we remove our assumptions, $\sum_{v\in\mathbb{A}(K)} pl(v) + dl(v) + \psi(v)$ increases by more than $(1/3 - \beta/3)t$ per doubly covered edge $vu \in E_{\mathbb{A}(K)}$ with $w_{vu} < (2/3 + \beta/3)t$, and decreases by at most $t - w_{vu}$ per uncovered edge $vu \in E_{\mathbb{A}(K)}$, plus possibly another t for the special node v^*. Hence, if we prove inequality (4.4) under the aforementioned assumptions, (4.3) must hold after we remove the assumptions, and Lemma 4.4.4 would follow.

We now turn to proving (4.4) when every covered edge $vu \in E_{\mathbb{A}(K)}$ has weight $w_{vu} = t$, and $\psi(v) = t$ for all $v \in \mathbb{A}(K)$. To this end, we consider the value $pl(v) + dl(v) + \psi(v)$ as a *budget* of node v. Furthermore, we also assign budgets

to edges $vu \in E_{\mathbb{A}(K)}$ that are doubly covered and have weight $w_{vu} < (2/3 + \beta/3)t$. Each of them gets a budget of $(1/3 - \beta/3)t$. Other remaining edges of $E_{\mathbb{A}(K)}$ have budget 0.

By redistributing budgets between nodes and edges, we will ensure that eventually

(i) every node in $\mathbb{A}(K)$ has a budget of at least t,

(ii) there exists a leaf in $\mathbb{A}(K)$ with budget strictly greater than $t + (2/3 + \beta/3)t$,

(iii) there exists a root in $\mathbb{A}(K)$ with budget at least $t + (2/3 - 2/3 \cdot \beta)t$,

(iv) every uncovered edge $vu \in E_{\mathbb{A}(K)}$ has a budget of at least $t - w_{vu}$, and

(v) no edge in $E_{\mathbb{A}(K)}$ has negative budget.

This would complete the proof.

We start with the leaf nodes. If $v \in \mathbb{A}(K)$ is a leaf, then (using Claim 4.4.11) it has a budget of more than $(5/3 + \beta/3)t - w_{vf(v)} + \psi(v) = (5/3 + \beta/3)t$. Using Claim 4.4.17, we can therefore add $(|\mathcal{D}(u)| - 1) \cdot (2/3 + \beta/3)t$ to the budget of every non-leaf $u \in \mathbb{A}(K)$, such that (i) and (ii) are still satisfied for all leaves.

Next we consider the roots. If $v \in \mathbb{A}(K)$ is a root and $|\mathcal{D}(v)| = 1$, then (using Claim 4.4.12) it has a budget of more than $(5/3 - 2/3 \cdot \beta)t$. If $v \in \mathbb{A}(K)$ is a root and $|\mathcal{D}(v)| > 1$, then (using Claim 4.4.13 and the load added in the previous step) it has a budget of at least $t + (|\mathcal{D}(v)| - 1) \cdot (2/3 + \beta/3)t$. In the latter case, we transfer $(2/3 - 2/3 \cdot \beta)t$ to the budget of every edge in $E_{\mathbb{A}(K)}$ that is incident with v. The budget of v thereby remains at least $t + (|\mathcal{D}(v)| - 1) \cdot (2/3 + \beta/3)t - |\mathcal{D}(v)| \cdot (2/3 - 2/3 \cdot \beta)t = (1/3 - \beta/3)t + |\mathcal{D}(v)| \cdot \beta t \geq (5/3 - 2/3 \cdot \beta)t$, where the last inequality follows from $|\mathcal{D}(v)| \geq 2$ and $\beta \geq 4/7$. Using Claim 4.4.17, we can thus add $(|\mathcal{F}(u)| - 1) \cdot (2/3 - 2/3 \cdot \beta)t$ to the budget of every non-root $u \in \mathbb{A}(K)$, such that (i) and (iii) are satisfied for all roots.

Before we move on to Type 1, 2, and 3 nodes, we take one step back and visit the leaves again, as their budget has increased again through the latest redistribution of load. Namely, every leaf $v \in \mathbb{A}(K)$ got an additional load of $(|\mathcal{F}(v)| - 1) \cdot (2/3 - 2/3 \cdot \beta)t$, which we now use to add $(2/3 - 2/3 \cdot \beta)t$ to the budget of every edge in $E_{\mathbb{A}(K)}$ that is incident with v, except to $vf(v)$ (which is surely covered). After this, (ii) and (iii) are satisfied, (i) holds for every root and every leaf, and every uncovered edge $vu \in E_{\mathbb{A}(K)}$ that is incident with a root or a leaf has a budget of at least $(2/3 - 2/3 \cdot \beta)t$.

Let us now consider the nodes of Type 1. Such a node v (using Claim 4.4.14 and the load added in previous steps) has a budget of at least $t + (|\mathcal{D}(v)| - 1) \cdot (2/3 + \beta/3)t + (|\mathcal{F}(v)| - 1) \cdot (2/3 - 2/3 \cdot \beta)t$. We transfer $(2/3 - 2/3 \cdot \beta)t$ to the budget of every edge in $E_{\mathbb{A}(K)}$ that is incident with v. Since there are $|\mathcal{D}(v)| + |\mathcal{F}(v)|$ such edges, the budget at v remains at least $t + (|\mathcal{D}(v)| - 1) \cdot (2/3 + \beta/3)t - (|\mathcal{D}(v)| + 1) \cdot (2/3 - 2/3 \cdot \beta)t = (|\mathcal{D}(v)| + 1)\beta t - (1/3 + 2/3 \cdot \beta)t \geq t$, as $|\mathcal{D}(v)| \geq 2$ and $\beta \geq 4/7$.

Next we consider the nodes of Type 2. Such a node v (using Claim 4.4.15 and the load added in previous steps) has a budget of more than $(2/3 + \beta/3)t + (|\mathcal{F}(v)| -$

$1) \cdot (2/3 - 2/3 \cdot \beta)t$. We transfer $(2/3 - 2/3 \cdot \beta)t$ to the budget of every edge in $E_{\mathbb{A}(K)}$ that is incident with v, except to $vf(v)$ and $vd(v)$ (which are surely covered). Since there are $|\mathcal{F}(v)| - 1$ such edges, the resulting budget at v is still more than $(2/3 + \beta/3)t$. We now reduce the budget of the edge $va(v)$ by $(1/3 - \beta/3)t$ and add this load to v's budget, which is then more than t. We will show later that this last step (reducing the budget of $va(v)$) does not cause a violation of (v).

Finally, we consider the nodes of Type 3. Such a node v (using Claim 4.4.16 and the load added in previous steps) has a budget of more than $(5/3 - 2/3 \cdot \beta)t + (|\mathcal{F}(v)| - 1) \cdot (2/3 - 2/3 \cdot \beta)t$. We transfer $(2/3 - 2/3 \cdot \beta)t$ to the budget of every edge in $E_{\mathbb{A}(K)}$ that is incident with v, except to $vn(v)$ (which is surely covered). Since there are $|\mathcal{F}(v)|$ such edges, the resulting budget at v is still more than t.

After the above redistributions of load, (i), (ii), and (iii) are satisfied. Furthermore, suppose that some edge $vu \in E_{\mathbb{A}(K)}$ is uncovered and has weight $w_{vu} \geq (2/3 + \beta/3)t$. Then at least once, we have added $(2/3 - 2/3 \cdot \beta)t$ to the budget of this edge, and we never reduced it. Therefore it has a budget of at least $(2/3 - 2/3 \cdot \beta)t \geq (1/3 - \beta/3)t \geq t - w_{vu}$, and (iv) holds for this edge. If, on the other hand, an uncovered edge $vu \in E_{\mathbb{A}(K)}$ has weight $w_{vu} < (2/3 + \beta/3)t$, then both u and v are in $\mathbb{A}(K)$ (due to activation rule 2), and $(2/3 - 2/3 \cdot \beta)t$ was added twice to the budget of vu. Furthermore, if this budget got reduced at some point, then at most once ($u = a(v)$ and $v = a(u)$ cannot happen simultaneously). The final budget of vu is thus at least $2 \cdot (2/3 - 2/3 \cdot \beta)t - (1/3 - \beta/3)t = t - \beta t > t - w_{vu}$. Hence, for such an edge the assertion (iv) also holds.

Finally, for (v), observe that the only point where we reduce the budget of a covered edge $vu \in E_{\mathbb{A}(K)}$ and add it to v's budget, is when v is of Type 2, $w_{vu} < (2/3 + \beta/3)t$, and $u = a(v)$. Furthermore, both u and v have to be in $\mathbb{A}(K)$ (due to activation rule 2). In this case, the budget of vu is reduced exactly once, by a value of $(1/3-\beta/3)t$. If vu is doubly covered, then it had an initial budget of $(1/3-\beta/3)t$, and its budget therefore remains non-negative. If, on the other hand, vu is covered but not doubly covered, then at some point its budget was increased by $(2/3 - 2/3 \cdot \beta)t$. Hence, the final budget is at least $(2/3 - 2/3 \cdot \beta)t - (1/3 - \beta/3)t = (1/3 - \beta/3)t \geq 0$. This concludes the proof.

Case 2: K is a cycle.

Claim 4.4.18. *K contains $\sum_{v \in K : \mathcal{F}(v) \neq \emptyset} (|\mathcal{F}(v)| - 1)$ many roots, and $\sum_{v \in K : \mathcal{D}(v) \neq \emptyset} (|\mathcal{D}(v)| - 1)$ many leaves. Furthermore, every root and leaf in K is activated.*

Proof. The first part simply follows from the degree sum formula for directed graphs and the fact that K is a cycle. The second part is analogous to Claim 4.4.17. □

We will again assume that every covered edge $vu \in E_{\mathbb{A}(K)}$ has weight $w_{vu} = t$,

and that $\psi(v) = t$ for all $v \in \mathbb{A}(K)$. With these assumptions, we will show that

$$\sum_{v\in\mathbb{A}(K)} pl(v) + dl(v) + \psi(v) > |\mathbb{A}(K)|t$$
$$- |\{\text{doubly covered } vu \in E_{\mathbb{A}(K)} : w_{vu} < (2/3 + \beta/3)t\}| \cdot (1/3 - \beta/3)t$$
$$+ |\{\text{uncovered } vu \in E_{\mathbb{A}(K)}\}| \cdot (t - w_{vu}). \tag{4.5}$$

By the same arguments as in Case 1, the error caused by the above two assumptions when we lower-bound the term $\sum_{v\in\mathbb{A}(K)} pl(v) + dl(v) + \psi(v)$ is:

- we underestimate the term by more than $(1/3-\beta/3)t$ per doubly covered edge $vu \in E_{\mathbb{A}(K)}$ with $w_{vu} < (2/3 + \beta/3)t$,
- we overestimate the term by at most $t - w_{vu}$ per uncovered edge $vu \in E_{\mathbb{A}(K)}$.

Note that, since K is a cycle, ψ must assign an edge from $E_{\mathbb{A}(K)}$ to every node in $\mathbb{A}(K)$, and thus there is no special node v^* as in Case 1. Hence, if we prove inequality (4.5) under the aforementioned assumptions, (4.3) must hold after we remove the assumptions, and Lemma 4.4.4 would follow.

We now prove (4.5) when every covered edge $vu \in E_{\mathbb{A}(K)}$ has weight $w_{vu} = t$, and $\psi(v) = t$ for all $v \in \mathbb{A}(K)$. Again, we consider the value $pl(v) + dl(v) + \psi(v)$ as a *budget* of node v. Furthermore, we also assign budgets to edges $vu \in E_{\mathbb{A}(K)}$ that are doubly covered and have weight $w_{vu} < (2/3+\beta/3)t$. Each of them gets a budget of $(1/3 - \beta/3)t$. Other remaining edges of $E_{\mathbb{A}(K)}$ have budget 0.

By redistributing budgets between nodes and edges, we will ensure that eventually

(i) every node in $\mathbb{A}(K)$ has a budget of at least t,

(ii) at least one node in $\mathbb{A}(K)$ has a budget strictly greater than t,

(iii) every uncovered edge $vu \in E_{\mathbb{A}(K)}$ has a budget of at least $t - w_{vu}$, and

(iv) no edge in $E_{\mathbb{A}(K)}$ has negative budget.

This would complete the proof.

We start with the leaf nodes. If $v \in \mathbb{A}(K)$ is a leaf, then (using Claim 4.4.11) it has a budget of more than $(5/3 + \beta/3)t - w_{vf(v)} + \psi(v) = (5/3 + \beta/3)t$. Using Claim 4.4.18, we can therefore add $(|\mathcal{D}(u)| - 1) \cdot (2/3+\beta/3)t$ to the budget of every non-leaf $u \in \mathbb{A}(K)$, such that (i) is still satisfied for all leaves.

Next we consider the roots. If $v \in \mathbb{A}(K)$ is a root and $|\mathcal{D}(v)| = 1$, then (using Claim 4.4.12) it has a budget of more than $(5/3 - 2/3 \cdot \beta)t$. If $v \in \mathbb{A}(K)$ is a root and $|\mathcal{D}(v)| > 1$, then (using Claim 4.4.13 and the load added in the previous step) it has a budget of at least $t + (|\mathcal{D}(v)| - 1) \cdot (2/3 + \beta/3)t$. In the latter case, we transfer $(2/3 - 2/3 \cdot \beta)t$ to the budget of every edge in $E_{\mathbb{A}(K)}$ that is incident with v. The budget of v thereby remains at least $t + (|\mathcal{D}(v)| - 1) \cdot (2/3+\beta/3)t - |\mathcal{D}(v)| \cdot$

$(2/3 - 2/3 \cdot \beta)t = (1/3 - \beta/3)t + |\mathcal{D}(v)| \cdot \beta t \geq (5/3 - 2/3 \cdot \beta)t$, where the last inequality follows from $|\mathcal{D}(v)| \geq 2$ and $\beta \geq 4/7$. Using Claim 4.4.18, we can thus add $(|\mathcal{F}(u)| - 1) \cdot (2/3 - 2/3 \cdot \beta)t$ to the budget of every non-root $u \in \mathbb{A}(K)$, such that (i) is satisfied for all roots.

Before we move on to Type 1, 2, and 3 nodes, we take one step back and visit the leaves again, as their budget has increased again through the latest redistribution of load. Namely, every leaf $v \in \mathbb{A}(K)$ got an additional load of $(|\mathcal{F}(v)| - 1) \cdot (2/3 - 2/3 \cdot \beta)t$, which we now use to add $(2/3 - 2/3 \cdot \beta)t$ to the budget of every edge in $E_{\mathbb{A}(K)}$ that is incident with v, except to $vf(v)$ (which is surely covered). After this, (i) holds for every root and every leaf, and every uncovered edge $vu \in E_{\mathbb{A}(K)}$ that is incident with a root or a leaf has a budget of at least $(2/3 - 2/3 \cdot \beta)t$.

As K is a cycle, there cannot be a node of Type 1, since every $v \in \mathbb{A}(K)$ with $|\mathcal{D}(v)| > 1$ is a root.

Let us now consider the nodes of Type 2. Such a node v (using Claim 4.4.15 and the load added in previous steps) has a budget of more than $(2/3+\beta/3)t+(|\mathcal{F}(v)|-1) \cdot (2/3 - 2/3 \cdot \beta)t$. We transfer $(2/3 - 2/3 \cdot \beta)t$ to the budget of every edge in $E_{\mathbb{A}(K)}$ that is incident with v, except to $vf(v)$ and $vd(v)$ (which are surely covered). Since there are $|\mathcal{F}(v)| - 1$ such edges, the resulting budget at v is still more than $(2/3 + \beta/3)t$. We now reduce the budget of the edge $va(v)$ by $(1/3 - \beta/3)t$ and add this load to v's budget, which is then more than t. We will show later that this last step (reducing the budget of $va(v)$) does not cause a violation of (iv).

Finally, we consider the nodes of Type 3. Such a node v (using Claim 4.4.16 and the load added in previous steps) has a budget of more than $(5/3 - 2/3 \cdot \beta)t + (|\mathcal{F}(v)|-1) \cdot (2/3-2/3 \cdot \beta)t$. We transfer $(2/3-2/3 \cdot \beta)t$ to the budget of every edge in $E_{\mathbb{A}(K)}$ that is incident with v, except to $vn(v)$ (which is surely covered). Since there are $|\mathcal{F}(v)|$ such edges, the resulting budget at v is still more than t.

After the above redistributions of load, (i) is satisfied. Furthermore, (ii) holds as at least one node must be of Type 2, Type 3, or a leaf, and for all these cases the load-inequality is a strict inequality. Finally, the proof of (iii) and (iv) is exactly analogous to the proof of (iv) and (v) in Case 1.

4.4.3. Proof of Lemma 4.4.5

In the following, let $E(V')$ denote the set of edges both of whose endpoints are in V' and $\delta(V')$ the set of edges exactly one of whose endpoints is in V', for each $V' \subseteq V$.

We prove the lemma by the following two steps.

Step 1: We create a clone of the pebble that is pushed from u^* to v^* and put this cloned pebble at v^* (by cloning, we mean the new pebble has the same weight and the same set of machines it can be assigned to) and keep the old one at u^*. We apply Explore2 to this new instance and argue that the outcome is "essentially the same" as if the cloned pebble were not there. More precisely, we show

Lemma 4.4.19. *Suppose that* Explore2 *is applied to the original instance (* before Push*) and the new instance with the cloned pebble at* v^**. Then at the end of each*

round i, $\mathbb{A}_i = \mathbb{A}_i^\dagger$ and $\mathbb{C}_i = \mathbb{C}_i^\dagger$, where $\mathbb{A}_i$, $\mathbb{A}_i^\dagger$ are the activated sets in the original and the new instances respectively, and $\mathbb{C}_i$ and $\mathbb{C}_i^\dagger$ are the conflict sets in the original and the new instances respectively.

Step 2: We then remove the original pebble at u^* but keep the clone at v^* (the same as the original instance after PUSH). Reapplying EXPLORE2, we then show that in each round, the set of activated nodes and the conflict set cannot enlarge. To be precise, we show[5]

Lemma 4.4.20. *Suppose that* EXPLORE2 *is applied to the new instance with the cloned pebble put at v^* and the original instance (*after PUSH*). Then at the end of each round i,*

1. $\bigcup_{\tau=0}^{i} \mathbb{A}'_\tau \subseteq \bigcup_{\tau=0}^{i} \mathbb{A}_\tau^\dagger$;
2. $\bigcup_{\tau=0}^{i} \mathbb{C}'_\tau \subseteq \bigcup_{\tau=0}^{i} \mathbb{C}_\tau^\dagger$;
3. *An edge not in $E(\bigcup_{\tau=0}^{i} \mathbb{C}_\tau^\dagger)$, if oriented in the original instance (after* PUSH*), must have the same orientation as in the new instance.*

Here $\mathbb{A}_i^\dagger$, $\mathbb{A}'_i$ are the activated sets in the new and the original instance (after PUSH*), respectively, and $\mathbb{C}_i^\dagger$ and $\mathbb{C}'_i$ are the conflict sets in the new and the original instances (after* PUSH*), respectively.*

Lemma 4.4.19 and Lemma 4.4.20(1) together imply Lemma 4.4.5, and the rest of this section is devoted to the proof of these two lemmas. First, however, we need to prove the following auxiliary lemma. It states that the "non-determinism" in the order of fake orientations does not matter, allowing us to let the two instances "mimic" the behavior of each other when we compare the conflict sets in the main proofs.

Lemma 4.4.21. *In the sub-procedure* Conflict set construction*, independent of the order of the edges being directed away from the new conflict set $\mathbb{C}_i$, the final outcome is the same in the following sense.*

1. *The sets of nodes in $\mathbb{C}_i$ is the same.*
2. *Every edge not in $E(\mathbb{C}_i)$ has the same orientation.*

Proof of Lemma 4.4.21

We plan to break each system into a set of subsystems and use the following lemma recursively to prove the lemma.

[5] Note that here we still refer to the instance with the cloned pebble at v^* as the *new* instance.

Lemma 4.4.22. *Let T be a tree of neutral edges in the beginning of the sub-procedure* Conflict set construction *whose nodes are all in $V \setminus \bigcup_{\tau=0}^{i-1} \mathbb{C}_\tau$ and consist of only the following two types:*

1. *Type α: a node v that (1) is already in $\mathbb{C}_i$ or has a directed path to a node in $\mathbb{C}_i$ in the beginning of the sub-procedure, or (2) at the end of all possible executions of the sub-procedure, it always has a directed path to some node in $\mathbb{C}_i \setminus T$.*
2. *Type β: a node v that (1) is not in $\mathbb{C}_i$ and does not have a directed path to a node in $\mathbb{C}_i$ in the beginning of the sub-procedure, and (2) at the end of all possible executions of the sub-procedure, it never has a directed path to some node in $\mathbb{C}_i$ via edges not in T. Furthermore, (3) all its incident neutral edges in the beginning of the sub-procedure are either in T, or never become directed towards v in any execution.*

Then the two properties of Lemma 4.4.21 hold. Namely, at the end of any execution, the final set $\mathbb{C}_i \cap T$ is the same and every edge in $T \setminus E(\mathbb{C}_i)$ has the same orientation.

Intuitively, Type α nodes in T are those bound to be part of $\mathbb{C}_i$ in any execution, while Type β nodes may or may not become part of $\mathbb{C}_i$. If a Type β node does become part of $\mathbb{C}_i$, then it must have a directed path to some Type α node in T via the edges in T after the execution. Notice also that by definition, a Type β node cannot be overloaded (otherwise, it is part of $\mathbb{A}_0 \subseteq \mathbb{C}_0$).

Proof. Let us first observe the outcome of an arbitrary execution of this sub-procedure. There can be two possibilities.

- **Case 1**. The entire tree T ends up being part of $\mathbb{C}_i$.
- **Case 2**. A set of sub-trees $T_1, T_2, \cdots$ become part of $\mathbb{C}_i$. The remaining nodes $T \setminus \bigcup_j T_j = \overline{F}$ form a forest. Each node $v \in \overline{F}$, if it has a non-$\overline{F}$ neighbor in T, then this neighbor is in some tree $T_j \subseteq \mathbb{C}_i$ and their shared edge is directed toward v.

The following claim is easy to verify and useful for our proof.

Claim 4.4.23. *Let $v \in T$ be a Type β node, and suppose that v has an incident edge in T that becomes outgoing during the execution of the sub-procedure. Then one of its incident edges in T must become incoming first, and furthermore $dl(v) + pl(v) + rl^i(v) + \sum_{u:vu \in T} w_{vu} > (5/3 + \beta/3)t$, where $rl^i(v)$ is the rock load of v in the beginning of the sub-procedure.*

We now consider the two cases separately.

Case 1: Suppose that in a different execution, the outcome is Case 2, i.e., there remains a forest $\overline{F} \subseteq T$ not being part of $\mathbb{C}_i$.

Choose a tree $\overline{T}$ in $\overline{F}$ and then choose any node in $\overline{T}$ as the root $\overline{r}$. Define the level of a node in $\overline{T}$ as its distance to $\overline{r}$. Consider the set of nodes v with the largest level l: they must be leaves of $\overline{T}$. By Proposition 4.4.2(3), in the new execution, all non-$\overline{F}$ neighbors of v in T direct their incident edges connecting v towards v. As a result, by Claim 4.4.23 and the fact that v becomes part of $\mathbb{C}_i$ in the original execution, v of level l must direct its incident edge in $\overline{T}$ toward its neighbor of level $l-1$ in $\overline{T}$. Nodes of level $l-1$ then have incoming edges from their neighbors of level l and from their non-$\overline{F}$ neighbors in T. So again they direct the edges in $\overline{T}$ towards the nodes of level $l-2$ in $\overline{T}$. Repeating this argument, we conclude that $\overline{r}$ receives all its incident edges in T in the new execution, a contradiction to Claim 4.4.23. This proves Case 1.

Case 2: Let us divide the incident edges in T of a node $v \in \overline{F}$ into three categories according to the outcome of the original execution: incoming $E_{\mathsf{i}}(v)$, outgoing $E_{\mathsf{o}}(v)$, and neutral $E_{\mathsf{n}}(v)$. Notice that by Proposition 4.4.2(3), all edges connecting v to its non-$\overline{F}$ neighbors in T are in $E_{\mathsf{i}}(v)$. Moreover, the following facts should be clear: at the end of any other execution, (1) an edge $e \in E_{\mathsf{o}}(v)$ must be directed away from v if all edges in $E_{\mathsf{i}}(v)$ are directed towards v, and (2) an edge in $E_{\mathsf{i}}(v) \cup E_{\mathsf{n}}(v)$ can be directed away from v only if beforehand some edge in $E_{\mathsf{o}}(v) \cup E_{\mathsf{n}}(v)$ is directed towards v, or v ends up being part of $\mathbb{C}_i$.

Claim 4.4.24. *Let $\overline{F} \subseteq T$ be the forest not becoming part of $\mathbb{C}_i$ in the original execution. In any other execution of the sub-procedure,*

1. *given $v \in \overline{F}$, it never happens that an edge $e \in E_{\mathsf{o}}(v) \cup E_{\mathsf{n}}(v)$ is directed towards v or an edge in $E_{\mathsf{i}}(v)$ is directed away from v;*

2. *none of the nodes in $\overline{F}$ ever becomes part of $\mathbb{C}_i$.*

Proof. Suppose that (2) is false and $v \in \overline{F}$ is the first node becoming part of $\mathbb{C}_i$. Then some edge $e = v_0u \in E_{\mathsf{i}}(v_0)$, where v_0 and v are connected in $\overline{F}$ and $u \in T$ is a non-$\overline{F}$ neighbor of v_0, is directed towards u beforehand. So (1) must be false first. Let $e' = v'u'$ be the first edge violating (1). (At this point, no node in $\overline{F}$ is part of $\mathbb{C}_i$ yet). If $e' \in E_{\mathsf{o}}(v') \cup E_{\mathsf{n}}(v')$ is directed toward v', then node u' directs edge e' towards v' because it first has another edge $e'' \in E_{\mathsf{o}}(u') \cup E_{\mathsf{n}}(u')$ coming toward itself. Then e'' should be the edge chosen, a contradiction. If $e' \in E_{\mathsf{i}}(v')$ is directed away from v', then some edge $e'' \in E_{\mathsf{o}}(v') \cup E_{\mathsf{n}}(v')$ is directed toward v' first, again implying that e'' should be chosen instead, another contradiction. Thus (1) and (2) hold. □

Claim 4.4.25. *Suppose that $T_j \subseteq \mathbb{C}_i$ in the original execution. Then in any other execution,*

1. *$T_j \subseteq \mathbb{C}_i$;*

2. Every edge $e = vu$ with $v \in T_j$ and $u \in \overline{F}$ is directed toward u.

Proof. For (1), we argue that T_j itself satisfies the condition of Lemma 4.4.22 and is exactly Case 1. For this, we need to show that a Type β node v of T in T_j is also a Type β node in T_j, i.e., v never has a directed path to some node in $\mathbb{C}_i$ via edges not in T_j. As v is a Type β node in T, it suffices to show that it cannot have a directed path to some Type α node in $T \backslash T_j$ via edges in T. Suppose there is such a path P. Then P must go through some node $u \in \overline{F}$, implying that u becomes part of $\mathbb{C}_i$ in this execution, a contradiction to Claim 4.4.24(2). This proves (1). (2) follows from Claim 4.4.24(2) and Proposition 4.4.2(3). □

What remains to be done is to show that all edges in $\overline{F}$ have the same orientation in any other execution. Let $L_0 \subseteq \overline{F}$ be the set of nodes v satisfying $|E_{\mathfrak{i}}(v) \cap \overline{F}| = 0$ and $L_{i>0} \subseteq \overline{F}$ be the set of nodes which can be reached from a node in L_0 by a directed path in $\overline{F}$ of maximum length exactly i after the original execution. In any other execution, by Claim 4.4.25(2), given $v \in L_0$, all edges in $E_{\mathfrak{i}}(v)$ are directed towards v, so all edges in $E_{\mathfrak{o}}(v) \cap \overline{F}$ are directed away from v. Now an inductive argument on i, combined with Claim 4.4.24(1), completes the proof of Case 2.

□

Proof. (of Lemma 4.4.21) We now explain how to make use of Lemma 4.4.22 to prove Lemma 4.4.21. For this, we decompose each system into a set of subsystems that satisfy the conditions required in Lemma 4.4.22.

First consider a system that is not a cycle. In the beginning of the sub-procedure *Conflict set construction*, let F be the forest consisting of the nodes in $V \backslash \bigcup_{\tau=0}^{i-1} \mathbb{C}_\tau$ and the edges that are neutral. We can assume that all nodes having a directed path to $\mathbb{C}_i$ are (already) in $\mathbb{C}_i$ as well.

Create a graph H whose node set are the connected components (trees) of F. If a non-$\mathbb{C}_i$ node in such a tree has a directed edge (we refer to the beginning of the sub-procedure) to some other non-$\mathbb{C}_i$ node in another tree, draw an arc from the node representing the former tree to the node representing the latter tree in H. (Intuitively, an arc in H indicates the possibility that a node in the former tree becomes part of $\mathbb{C}_i$ because of a directed edge to a node in $\mathbb{C}_i$ in the latter tree). As the entire system is not a cycle, some node in H must have out-degree 0. It is easy to verify that the particular tree corresponding to this node satisfies the conditions in Lemma 4.4.22, so the lemma can be applied to it.

We now find the next tree satisfying the conditions of Lemma 4.4.22 by redefining the graph H as follows. Observe that the "processed" tree (the one we applied Lemma 4.4.22 to) has exactly two types of non-$\mathbb{C}_i$ nodes in the beginning of the sub-procedure: those that always become part of $\mathbb{C}_i$ (i.e., in every possible execution of the sub-procedure) and those that never become part of $\mathbb{C}_i$. Nodes in other trees that, in the beginning, have a directed edge to the former type of nodes are bound to become part of $\mathbb{C}_i$ (i.e., they satisfy the conditions of a Type α node in their tree). Nodes in other trees with a directed edge to the latter type of nodes are not to

become part of $\mathbb{C}_i$ because of them. So in H, we can just remove the corresponding arcs and the node representing the already processed tree. In the updated H, the node with out-degree 0 is the next tree, on which Lemma 4.4.22 can be applied. Repeating this procedure, we are done with the first case (when the system is not a cycle).

Finally, consider the case that the entire system is a cycle. For the special case that the entire cycle consists of neutral edges, it is easy to verify that Lemma 4.4.21 holds. So suppose that the set of neutral edges form a forest (precisely, a set of disjoint paths). We can proceed as before—build H and find a vertex in H with out-degree 0 and recurse—except for the special case that H is a directed cycle V_1, V_2,... in the beginning. Observe that the last node $v \in V_1$ has a directed edge to the first node $u \in V_2$ and neither v nor u is in $\mathbb{C}_i$. Similarly, the last node of V_2 is also not in $\mathbb{C}_i$ and neither is the first node of V_3 and so on. In this case, it is easy to see that Lemma 4.4.21 holds for the entire system. □

Proof of Lemma 4.4.19

When Explore2 is applied on the original instance before Push, suppose that v^* joins the conflict set in round k, i.e., $v^* \in \mathbb{C}_k$. We first make the following claim.

Claim 4.4.26. *Apply* Explore2 *to the new instance. In round k, immediately after the sub-procedure* Conflict set construction*, the following holds.*

1. $\mathbb{A}_\tau = \mathbb{A}^\dagger_\tau$*, for* $0 \le \tau \le k$*,*
2. $\mathbb{C}_\tau = \mathbb{C}^\dagger_\tau$*, for* $0 \le \tau \le k$*,*
3. *Edges not in* $E(\bigcup_{\tau=0}^{k} \mathbb{C}_\tau)$ *have the same orientations in both instances.*

We will prove the claim shortly after. In the following, we will show that $\mathbb{A}_k = \mathbb{A}^\dagger_k$ at the end of round k. Combining this with Claim 4.4.26(2)(3) and Lemma 4.4.21, an inductive argument proves that Lemma 4.4.19 is true also from round k onwards.

Recall that by the definition of Push, at the end of Explore2 in the original instance, either (1) $\mathcal{D}(v^*) = \emptyset$, or (2) $dl(v^*) + pl(v^*) + w_{v^*u} \le (5/3 - 2/3 \cdot \beta)t$ for all $u \in \mathcal{F}(v^*)$. We consider these two cases separately.

Case 1: Suppose that $\mathcal{D}(v^*) = \emptyset$ in the original instance at the end of Explore2. We will show that at the end of round k, $\mathbb{A}_k = \mathbb{A}^\dagger_k$ and in particular $v^* \in \mathbb{A}_k = \mathbb{A}^\dagger_k$. By Claim 4.4.26(1), we just have to argue that a node is activated by Rule 1 or Rule 2 in the original instance if and only if it is activated by one of these two rules in the new instance, in round k.

For v^*, recall that it is part of $\mathbb{C}_k$. It becomes so by either (1) being a Type A node in $\mathbb{A}_k$, or (2) having an outgoing edge v^*u and $u \in \bigcup_{\tau=0}^{k} \mathbb{C}_\tau$. For the former case, Claim 4.4.26(1) shows that $v^* \in \mathbb{A}^\dagger_k$. For the latter case, as $\mathcal{D}(v^*) = \emptyset$ at the end of Explore2 in the original instance, in round k, $dl(v^*)+pl(v^*)+w_{v^*u} > (5/3+\beta/3)t$, and hence Rule 1 applies to v^*. In the new instance, the preceding inequality still

holds since the pebble load of v^* is increased by the cloned pebble. As $u \in \bigcup_{\tau=0}^{k} \mathbb{C}_\tau^\dagger$ (Claim 4.4.26(2)), Rule 1 again applies to v^* (note that u is still a father of v^*, since otherwise v^* would be overloaded and part of both $\mathbb{A}_0^\dagger$ and $\mathbb{A}_0$).

For other nodes $v \neq v^*$, as $pl(v) + dl(v)$ are the same in both instances, if v is activated by Rule 1 in the original instance, then it is so too in the new instance, and vice versa. We have established that the set of nodes activated by Rule 1 is the same in both instances. Now by Claim 4.4.26(2), the set of nodes activated by Rule 2 is again the same in both instances. Therefore, $\mathbb{A}_k = \mathbb{A}_k^\dagger$ at the end of round k.

Case 2: Suppose that $dl(v^*) + pl(v^*) + w_{v^*u} \leq (5/3 - 2/3 \cdot \beta)t$ for all $u \in \mathcal{F}(v^*)$ in the original instance. Then v^* cannot be a Type B node in the original instance, i.e., it is not activated by Rule 1 (but it is possible that v^* is activated by Rule 2 or as a Type A node). We now argue that in the new instance, in round k, v^* cannot be activated by Rule 1 either.

By the definition of Push (specifically Definition 4.4.3(3)(4)), in the original instance, each father and child $u \in \bigcup_{\tau=0}^{k} \mathbb{C}_\tau$ of v^* satisfies $dl(v^*) + pl(v^*) + w_{v^*u} \leq (5/3 - 2/3\beta)t$ (notice that when we compare original and new instance, a father can become a child and vice versa). Therefore, even with the cloned pebble (of weight at most βt) in the new instance, Rule 1 still cannot be applied to v^* in round k.

For other nodes $v \neq v^*$, it is easy to see that v is activated by Rule 1 in the original instance if and only if in the new instance in round k. We have established that the set of nodes activated by Rule 1 is the same in both instances. Now by Claim 4.4.26(2), the set of nodes activated by Rule 2 is again the same in both instances. Therefore, $\mathbb{A}_k = \mathbb{A}_k^\dagger$ at the end of round k.

Proof of Claim 4.4.26: Consider the moment at the end of round $k-1$ when Explore2 is applied on the original instance before Push. In the special case of $k = 0$, we refer to the moment immediately after Forced Orientations is called in the initialization of Explore2.

In this moment, let us put the cloned pebble at v^* and invoke Forced Orientations. This causes a (possibly empty) set of neutral edges $\overline{E}$ to become directed. Let V_0 be the set of nodes which are the heads or tails of the now directed edges in $\overline{E}$. Let V_1 be the set of nodes that can be arrived at from nodes in V_0 following the other directed edges E^* (i.e., those that are already oriented at the end of round $k-1$ before the cloned pebble is put at v^*). Observe that $v^* \in V_0$ can reach any node in $V_0 \cup V_1$ by following the directed edges in $\overline{E} \cup E^*$. Let $E_{\mathfrak{i}}(v)$, $E_{\mathfrak{o}}(v)$, and $E_{\mathfrak{n}}(v)$ denote the set of incident incoming, outgoing, neutral edges of each node $v \in V$ after we put the cloned pebble and called Forced Orientations. It should be clear that (1) $\overline{E} \subseteq \bigcup_{v \in V_0} E_{\mathfrak{o}}(v)$, (2) $\bigcup_{v \in V_0 \cup V_1} E_{\mathfrak{o}}(v) \cap \delta(V_0 \cup V_1) = \emptyset$, and (3) none of the nodes in V_0 is overloaded at the end of round $k-1$ (and hence also not in subsequent rounds).

Claim 4.4.27. *When* Explore2 *is applied on the original instance before* Push,

1. *If an edge e is in $\overline{E} \cap E_{\mathfrak{o}}(v)$ for some $v \in V_0$, then at the end of round k, edge e is also an outgoing edge of v (independent of the order of fake orientations);*

2. *At the end of round $k-1$, none of the nodes in $V_0 \cup V_1$ is part of the conflict set built so far, i.e.* $(V_0 \cup V_1) \cap \bigcup_{\tau=0}^{k-1} \mathbb{C}_\tau = \emptyset$.

Proof. Consider the edge $v^*u \in \overline{E} \cap E_{\mathfrak{o}}(v^*)$. As v^* is part of $\mathbb{C}_k$, at the end of round k, v^*u cannot be neutral. As it is directed toward u after the added cloned pebble,

$$w_{v^*u} + pl(v^*) + dl(v^*) + rl^{k-1}(v^*) + w > (5/3 + \beta/3)t, \tag{4.6}$$

where w is the weight of the cloned pebble and $rl^{k-1}(v^*)$ is the weight of the rocks assigned to v^* at the end of round $k-1$. Suppose for a contradiction that edge v^*u is directed toward v^* at the end of round k. Recall that by Definition 4.4.3(3), for the pebble to be pushed from u^* to v^* in the original instance, $pl(v^*) + dl(v^*) + rl(v^*) \leq (5/3 - 2/3 \cdot \beta)t$, where $rl(v^*)$ is the weight of the rocks assigned to v^* at the end of EXPLORE2. Then

$$dl(v^*) + pl(v^*) + (rl^{k-1}(v^*) + w_{v^*u}) + w \leq dl(v^*) + pl(v^*) + rl(v^*) + w \leq (5/3 + \beta/3)t,$$

a contradiction to inequality (4.6). So we establish that v^*u is directed toward u at the end of round k. Consider u and its incident edge $uu' \in \overline{E} \cap E_{\mathfrak{o}}(u)$. The fact that v^*u causes uu' to be directed toward u' implies that at the end of round k, uu' cannot be directed toward u or stay neutral. Repeating this argument, we prove (1).

If a node in $V_0 \cup V_1$ is part of $\bigcup_{\tau=0}^{k-1} \mathbb{C}_\tau$, then either v^* is part of $\bigcup_{\tau=0}^{k-1} \mathbb{C}_\tau$, a contradiction to the assumption that v^* joins the conflict set in round k, or some node in $V_0 \backslash \{v^*\}$ has an incident edge in $\overline{E}$ directed away from it at the end of round $k-1$ (see Proposition 4.4.2(3)), a contradiction to the definition of $\overline{E}$. This proves (2).

□

Claim 4.4.27(2) has the important implication that, in the original instance, the set of nodes $V_0 \cup V_1$ is "isolated" from the rest of the graph up to the end of round $k-1$ in EXPLORE2: they do not have a directed path to nodes in $\bigcup_{\tau=0}^{k-1} \mathbb{C}_\tau$ and they are not reachable from nodes in $\bigcup_{\tau=0}^{k-2} \mathbb{A}_\tau$.

Claim 4.4.28. *Suppose that $k \geq 1$. When* EXPLORE2 *is applied on the new instance, at the end of round $k-1$,*

1. *Every edge $e \in E_{\mathfrak{o}}(v)$ (respectively $E_{\mathfrak{i}}(v)$, $E_{\mathfrak{n}}(v)$) for any $v \in V_0 \cup V_1$ is an outgoing (respectively incoming, neutral) edge of v in the new instance;*

2. $\mathbb{A}_\tau = \mathbb{A}_\tau^\dagger$ *for* $0 \leq \tau \leq k-1$;

3. $\mathbb{C}_\tau = \mathbb{C}_\tau^\dagger$, *for* $0 \leq \tau \leq k-1$;

4. *Every edge not in* $E(\bigcup_{\tau=0}^{k-1} \mathbb{C}_\tau) \cup \overline{E}$ *has the same orientation in both instances.*

Proof. By Claim 4.4.27(2), none of the nodes in $V_0 \cup V_1$ is overloaded in the original instance, as $k \geq 1$. By Lemma 4.4.21, we may assume that both instances decide their fake orientations based on the same fixed total order. Let us define the following events for both instances:

- β: An edge in $E(V_0 \cup V_1)$ becomes directed.
- α_1: An edge not in $E(V_0 \cup V_1)$ becomes directed.
- α_2: The sub-procedure *Activation of nodes* is executed.
- α_3: A new round starts and a set of (Type A-) nodes is activated.
- α_4: The internal while-loop of *Conflict set construction* is executed and a set of nodes is added into the conflict set.

Using an inductive argument, the following fact can easily be verified:

> As long as no edge in $E_\mathfrak{n}(v) \cup E_\mathfrak{i}(v)$ becomes outgoing for any $v \in V_0 \cup V_1$, the sequences of α-events are the same in both instances (but possibly intermitted by different sequences of β-events) up to the end of round $k-1$. Furthermore, right after two corresponding α-events in the original and new instance, the conflict set and activated nodes, and the direction of all edges not in $E(V_0 \cup V_1)$ are the same in both instances.

To prove (1), consider the first moment in the new instance when an edge $e \in E_\mathfrak{n}(v) \cup E_\mathfrak{i}(v)$ becomes outgoing for any $v \in V_0 \cup V_1$ before the end of round $k-1$. For this to happen, as v is not overloaded in the original instance, there exists another edge $e' = vu \in E_\mathfrak{n}(v) \cup E_\mathfrak{o}(v)$ becoming incoming for v first. By the above fact, it must be the case that $u \in V_0 \cup V_1$. Then $e' \in E_\mathfrak{n}(u) \cup E_\mathfrak{i}(u)$, and e' becomes outgoing for u before e becomes outgoing for v, a contradiction.

We next show that every edge $e \in E_\mathfrak{o}(v)$ is an outgoing edge for $v \in V_0 \cup V_1$ at the end of round $k-1$ in the new instance. Assume that v^*'s system is not a cycle. Then $E_\mathfrak{i}(v^*) \subseteq \delta(V_0 \cup V_1)$, and all these edges are incoming at the end of round $k-1$, implying that all edges in $E_\mathfrak{o}(v^*)$ must be outgoing. Now an inductive argument on the rest of the nodes $v \in V_0 \cup V_1$ (based on their distance to v^*) establishes that $e \in E_\mathfrak{o}(v)/E_\mathfrak{i}(v)/E_\mathfrak{n}(v)$ is an outgoing/incoming/neutral edge of v at the end of round $k-1$ in the new instance. The cycle-case follows by a similar argument. This completes the proof of (1).

Finally, combining (1) with the above fact, the rest of the claim follows. □

Claim 4.4.29. *Suppose that $k = 0$. When* Explore2 *is applied on the new instance, at the end of the initialization (after* Forced Orientations*),*

1. *Every edge $e \in E_\mathfrak{o}(v)$ (respectively $E_\mathfrak{i}(v)$, $E_\mathfrak{n}(v)$) for any $v \in V_0 \cup V_1$ is an outgoing (respectively incoming, neutral) edge of v in the new instance;*
2. *Every edge not in $E(V_0 \cup V_1)$ has the same orientation in both instances;*

3. The set of overloaded nodes are the same in both instances.

Proof. In the new instance, we claim that no edge in $E_{\mathfrak{n}}(v) \cup E_{\mathfrak{i}}(v)$ becomes outgoing for any $v \in V$ during the initialization. Suppose not and $e \in E_{\mathfrak{n}}(v) \cup E_{\mathfrak{i}}(v)$ is the first such edge. If this happens because another edge $e' = vu \in E_{\mathfrak{o}}(v) \cup E_{\mathfrak{n}}(u)$ is directed toward v first, then e' should have been chosen. So v must be overloaded in the original instance and by Lemma 4.4.1, e is the only edge in $E_{\mathfrak{i}}(v)$ and $dl(v) + pl(v) + w_{e=vu_0} > (5/3 + \beta/3)t$. (Notice that $v \neq v^*$).

Consider the moment in the initialization of the original instance, when $e = vu_0$ is directed toward v. First suppose that in this moment, u_0 has no incoming edges yet. Then we know that $dl(u_0) + pl(u_0) + w_{vu_0} > (5/3 + \beta/3)t$ and the pair (u_0, v) precedes (v, u_0) in the total order of edges. This is still true in the new instance, contradicting our assumption that vu_0 is chosen to be directed toward u_0. So u_0 already has some incoming edges E_{u_0} in the original instance. In the new instance, when vu_0 is directed toward u_0, it cannot be that all edges of E_{u_0} are already directed toward u_0. So at least one such $u_1u_0 \in E_{u_0}$ is still neutral (it cannot be outgoing because of the choice of $e = vu_0$). Repeating this argument, in the new instance, we find a path of neutral edges $vu_0u_1\ldots$ immediately before $e = vu_0$ is directed toward u_0, and this path ends at a node u_z where $dl(u_z) + pl(u_z) + w_{u_zu_{z-1}} > (5/3 + \beta/3)t$, and the pair (u_z, u_{z-1}) precedes the pair (v, u_0). This contradicts the assumption that $e = vu_0$ is chosen to be directed toward u_0.

So we established that no edge in $E_{\mathfrak{n}}(v) \cup E_{\mathfrak{i}}(v)$ becomes outgoing for any $v \in V$. To complete the proof of (1) and (2), suppose that $uv \in E_{\mathfrak{i}}(v)$ for some $v \in V$ remains neutral after the initialization of the new instance. Then there must be another edge $wu \in E_{\mathfrak{i}}(u)$ which also remains neutral. Repeating this argument, we conclude that the entire system is a cycle, whose edges are all neutral after the initialization of the new instance. As $uv \in E_{\mathfrak{i}}(v)$, there must be some edge xy in this cycle, so that $dl(x) + pl(x) + w_{xy} > (5/3 + \beta/3)t$ after the cloned pebble is put on v^*. This edge cannot remain neutral after the initialization of the new instance, a contradiction.

Finally, (3) follows from (1) and (2), and the fact that no node in V_0 is overloaded in both instances.

□

To complete the proof of Claim 4.4.26, we now show that in round k, after the sub-procedure *Conflict set construction*, the outcome of the two instances are exactly the same, except for the orientation of the edges in $E(\bigcup_{\tau=0}^{k} \mathbb{C}_\tau)$. Notice that by Claim 4.4.28(1)(4) and Claim 4.4.29(1)(2), at the end of round $k-1$, the orientations of all edges not in $E(\bigcup_{\tau=0}^{k-1} \mathbb{C}_\tau)$ are the same in both instances, with the only exception that $\overline{E}$ are oriented in the new instance but neutral in the original instance. Furthermore, by Claim 4.4.28(2) and Claim 4.4.29(3), the same set of nodes are added into $\mathbb{A}_k^\dagger$, $\mathbb{A}_k$, $\mathbb{C}_k^\dagger$, $\mathbb{C}_k$ in the beginning of round k (as Type A nodes).

Let $V_1' \subseteq V_1$ be the set of nodes that can be reached by a directed path from v^* in the original instance at the end of round $k-1$ (such a path does not use edges in $\overline{E}$).

Let us first suppose the system containing v^* is a tree. In the following, when we say the "sub-tree" of an edge $e \in E_{\mathfrak{n}}(v)$ for some $v \in V_0 \cup V_1$, we mean the sub-tree outside of $V_0 \cup V_1$ connected to $V_0 \cup V_1$ by the edge e (note that $e \in \delta(V_0 \cup V_1)$). We now make use of Lemma 4.4.21 to let the two instances mirror each other's behavior. Consider how v^* becomes part of $\mathbb{C}_k$ in the original instance.

Case 1: in the beginning of round k, v^* or some node in V_1' becomes a Type A node. Then v^* becomes part of the conflict set in both instances in the beginning of the sub-procedure *Conflict set construction*, before any further edges become directed. In this case, in the original instance, let v^* direct all edges in $\overline{E}$ away from v^* (by running ahead a few iterations and picking the respective edges incident with v^* as fake orientations). After that, in both instances, direct all remaining neutral edges incident with v^* away from v^*. Now the two instances are the same[6], and we can let them continue identically until the end of the sub-procedure (also note that all edges incident with v^* are already oriented in both instances).

Case 2: in the sub-procedure *Conflict set construction*, due to fake orientations in the sub-trees of edges in $\cup_{v \in V_1'} E_{\mathfrak{n}}(v)$, some nodes in V_1' (hence v^*) become part of $\mathbb{C}_k$. In this case, in both instances, apply these fake orientations first. Then v^* becomes part of the conflict set in both instances. Let the original instance direct the edges in $\overline{E}$ away from v^*, and then, in both instances, direct all remaining neutral edges incident with v^* away from v^*. Now the two instances are the same, and we can let them continue identically until the end of the sub-procedure.

Case 3: the above two cases do not apply. Consider the execution of the sub-procedure *Conflict set construction* in the original instance in round k. $E_{\mathfrak{n}}(v^*)$ can be partitioned into $E_{\mathfrak{n}\to\mathfrak{i}}(v^*)$ and $E_{\mathfrak{n}\to\mathfrak{o}}(v^*)$, the former (latter) being those edges in $E_{\mathfrak{n}}(v^*)$ becoming incoming (outgoing) inside the sub-procedure.

Observe that (1) $E_{\mathfrak{n}\to\mathfrak{i}}(v^*) \neq \emptyset$, otherwise v^* cannot become part of $\mathbb{C}_k$ in the original instance (see Claim 4.4.27(1)), and (2) in round k, as long as no edge $E_{\mathfrak{n}\to\mathfrak{o}}(v^*)$ is directed toward v^*, then even with the cloned pebble at v^*, a proper subset $E' \subset E_{\mathfrak{n}\to\mathfrak{i}}(v^*)$ directed toward v^* cannot cause another edge in $E_{\mathfrak{n}\to\mathfrak{i}}(v^*)\backslash E'$ to be directed away from v^* (by Definition 4.4.3(3) and the fact that $rl(v^*) = \sum_{e \in E_{\mathfrak{n}\to\mathfrak{i}}(v^*) \cup E_{\mathfrak{i}}(v^*)} w_e$ in the original instance after round k).

Let the original instance start round k with the fake orientations in the sub-trees of edges in $E_{\mathfrak{n}\to\mathfrak{i}}(v^*)$ until all edges in $E_{\mathfrak{n}\to\mathfrak{i}}(v^*)$ are directed toward v^*, and let the new instance mimic. Now the edges in $\overline{E}$ are directed away from v^* also in the original instance (since any rock edge is heavier than the cloned pebble). Hence, all edges not in $E(\bigcup_{\tau=0}^{k-1} \mathbb{C}_\tau)$ have the same orientations in both instances, except that possibly some edges in $E_{\mathfrak{n}\to\mathfrak{o}}(v)$ and in their sub-trees are already oriented in the new instance while not in the original instance (this is because in the new instance, the pebble load at v^* is higher). Let $E'_{\mathfrak{n}\to\mathfrak{o}} \subseteq E_{\mathfrak{n}\to\mathfrak{o}}(v^*)$ be those edges in $E_{\mathfrak{n}\to\mathfrak{o}}(v^*)$ that are already oriented in the original instance at this point. Now let the original instance apply all possible fake orientations in the sub-trees of edges

[6] When we say that two instances are *the same* at a certain time point, we mean that the conflict set and activated nodes, and the orientation of all edges not in $E(\bigcup_{\tau=0}^{k-1} \mathbb{C}_\tau)$ are the same.

in $\bigcup_{v\in V_0\cup V_1\setminus\{v^*\}} E_{\mathfrak{n}}(v) \cup E'_{\mathfrak{n}\to\mathfrak{o}}$ and let the new instance mimic. After this step, v^* must be part of the conflict set $\mathbb{C}_k$ and $\mathbb{C}^\dagger_k$ in both instances. Finally, in both instances, direct all remaining neutral edges in $E_{\mathfrak{n}\to\mathfrak{o}}(v^*)$ away from v^*. Now the two instances are the same, and we can let them continue identically until the end of the sub-procedure. This finishes the proof of the tree case.

Next suppose that the system containing v^* is a cycle. In the original instance, v^* joins $\mathbb{C}_k$ in two possible ways. Either v^* or some node in V'_1 is a Type A node (then this is the same as Case 1 above), or during the sub-procedure *Conflict set construction* an edge e_α is directed toward v^*, causing the other incident edge e_β to be directed away from v^*. In this case, by Definition 4.4.3(4), $\overline{E} = \emptyset$. Hence, the two instances are the same already in the beginning of the sub-procedure, and we can let them perform identically by choosing the same fake orientations. This finishes the cycle case and the entire proof of Claim 4.4.26.

Proof of Lemma 4.4.20

Our idea is to make use of Lemma 4.4.21: we will apply Explore2 simultaneously to both instances and let the new instance mimic the behavior of the original instance. In the following, we implicitly assume that nodes having a directed path to $\bigcup_{\tau=0}^{i} \mathbb{C}^\dagger_\tau$ (respectively $\bigcup_{\tau=0}^{i} \mathbb{C}'_\tau$) are part of it in the new (original) instance. Furthermore, at any time point considered, we refer to the current content of the sets $C^\dagger_i$ and $\mathbb{C}'_i$. The lemma below explains how the mimicking is done.

Lemma 4.4.30. *In round $i \geq 0$, suppose that both instances are in the sub-procedure* Conflict set construction *and Lemma 4.4.20(2)(3) hold. Let the original instance apply an arbitrary fake orientation and invoke* Forced Orientations. *Then the new instance can apply a number of fake orientations so that Lemma 4.4.20(2)(3) still hold.*

Proof. In the original instance, suppose that the chosen fake orientation is to direct the edge $e_0 = v_0u_0$ toward u_0. In the subsequent call of Forced Orientations, a tree T_{u_0} of neutral edges are further directed away from u_0. Given two incident edges e, e' of a node $v \in T_{u_0}$, we write $e \prec e'$ if e is closer to u_0 than e'. Similarly, given two adjacent nodes $v, u \in T_{u_0}$, we write $v \prec u$ if v is closer to u_0 than u. We make an important observation.

Claim 4.4.31. *Suppose that $v \in T_{u_0}$ and $v \notin \bigcup_{\tau=0}^{i} \mathbb{C}^\dagger_\tau$. Furthermore, suppose that $e, e' \in T_{u_0}$ are incident on v and $e \prec e'$. Then v can take at most one of them in the new instance, i.e., if either of them is directed toward v, then (after* Forced Orientations*) the other must be directed away from v.*

In the special case of $v = u_0 \notin \bigcup_{\tau=0}^{i} \mathbb{C}^\dagger_\tau$, assuming that e is an incident edge of u_0 in T_{u_0}, u_0 can take at most one of $e_0 = v_0u_0$ and e.

Proof. The dedicated load $dl(v)$ and the pebble load $pl(v)$ are at least as heavy in the new instance as in the original. An edge not in $E(\bigcup_{\tau=0}^{i} \mathbb{C}^\dagger_\tau)$, if oriented in the

original instance, must be oriented in the same way in the new instance. So the rock load $rl(v)$ is also at least as heavy in the new instance as in the original. Thus, if in the original instance, e being directed toward v causes e' to be directed away from v, then v can take at most one of them in the original, and hence in the new instance. The second part of the claim follows from the same reasoning.

□

Our goal is to apply a number of fake orientations in the new instance, so that the edges $(\{e_0\} \cup T_{u_0}) \backslash E(\bigcup_{\tau=0}^{i} \mathbb{C}_\tau^\dagger)$ are directed the same way as in the original instance.

First, if e_0 is still neutral in the new instance, direct it toward u_0 and invoke FORCED ORIENTATIONS. Notice that if e_0 is already directed toward v_0 in the new instance, then both $v_0, u_0 \in \bigcup_{\tau=0}^{i} \mathbb{C}_\tau^\dagger$, and hence $e_0 \in E(\bigcup_{\tau=0}^{i} \mathbb{C}_\tau^\dagger)$.

We make another observation.

Claim 4.4.32. *In the new instance, after a call of* FORCED ORIENTATIONS, *assume that* $e = vu \in T_{u_0}$, *and* $v \prec u$.

1. *If* $e = vu$ *is directed toward* v, *then both* $u, v \in \bigcup_{\tau=0}^{i} \mathbb{C}_\tau^\dagger$.
2. *If* $e = vu$ *is directed toward* $u \notin \bigcup_{\tau=0}^{i} \mathbb{C}_\tau^\dagger$, *then the entire sub-tree of* T_{u_0} *rooted at* u *is directed away from* u_0 *and none of its nodes is in* $\bigcup_{\tau=0}^{i} \mathbb{C}_\tau^\dagger$.

Proof. For (1), suppose that e is directed toward v. If $v \in \bigcup_{\tau=0}^{i} \mathbb{C}_\tau^\dagger$, then so is u and the claim holds. So assume that $v \notin \bigcup_{\tau=0}^{i} \mathbb{C}_\tau^\dagger$. Consider the incident edge $e' \in T_{u_0}$ of v with $e' \prec e$. By Claim 4.4.31, e' must also be directed toward u_0. Repeating this argument, we find a sequence of edges directed toward u_0 and they either end up at a node in $\bigcup_{\tau=0}^{i} \mathbb{C}_\tau^\dagger$ (then implying that v is part of $\bigcup_{\tau=0}^{i} \mathbb{C}_\tau^\dagger$, a contradiction), or at u_0 and $u_0 \notin \bigcup_{\tau=0}^{i} \mathbb{C}_\tau^\dagger$. Then, by Claim 4.4.31, the edge v_0u_0 must be directed toward v_0 in the new instance, again implying that v is part of $\bigcup_{\tau=0}^{i} \mathbb{C}_\tau^\dagger$, a contradiction.

(2) is the consequence of Claim 4.4.31 and our assumption that all nodes having a directed path to $\bigcup_{\tau=0}^{i} \mathbb{C}_\tau^\dagger$ are part of it. □

In the new instance, the set of neutral edges in T_{u_0} form a set of node-disjoint trees $T_1, T_2, \ldots$, where each tree T_j has a root node r_j that is closest to u_0 in T_{u_0} (r_j could be u_0 itself). Observe that no node in T_j can be part of $\bigcup_{\tau=0}^{i-1} \mathbb{C}_\tau^\dagger$, since otherwise its incident edges would not be neutral in round i. It follows from Claim 4.4.32 (resp. the last part of Claim 4.4.31 if $r_j = u_0$) that $r_j \in \mathbb{C}_i^\dagger$. Hence, we can let the new instance direct the neutral edges in T_j incident on r_j away from it. If some edge in T_j remains neutral after this, by Claim 4.4.31, there must exist a node $v \in T_j \cap \mathbb{C}_i^\dagger$ with neutral incident edges in T_j. Then again let v direct all remaining neutral edges in T_j away from it and continue this process until all edges in T_j are directed away from u_0.

By the above mimicking, we guarantee that all edges in $(\{e_0\} \cup T_{u_0}) \backslash E(\bigcup_{\tau=0}^{i} \mathbb{C}_\tau^\dagger)$ are directed the same way in both instances. This implies that Lemma 4.4.20(3)

holds after the mimicking. Next we argue that if a node v is added into $\mathbb{C}'_i$ in the original instance, then it is either already in $\bigcup_{\tau=0}^{i} \mathbb{C}^{\dagger}_{\tau}$, or is added into $\mathbb{C}^{\dagger}_i$ as well after the mimicking. For v to be added into $\mathbb{C}'_i$ in the original instance, it must have a directed path P to some node $\hat{v} \in \mathbb{C}'_i$ after v_0u_0 is oriented toward u_0, where $\hat{v}$ is part of $\mathbb{C}'_i$ already before the fake orientation. Note that $\hat{v}$ is also in $\bigcup_{\tau=0}^{i} \mathbb{C}^{\dagger}_{\tau}$ before the mimicking. Let $\overline{v}$ be the first node on P (starting from v) that is part of $\bigcup_{\tau=0}^{i} \mathbb{C}^{\dagger}_{\tau}$ after the mimicking. If $\overline{v} = v$, we are done. Otherwise, since Lemma 4.4.20(3) holds after the mimicking, $v \notin \bigcup_{\tau=0}^{i} \mathbb{C}^{\dagger}_{\tau}$ has a directed path to $\overline{v} \in \bigcup_{\tau=0}^{i} \mathbb{C}^{\dagger}_{\tau}$ in the new instance, a contradiction.

So we have established Lemma 4.4.20 (2) and (3) after the mimicking. □

We use the above lemma to prove Lemma 4.4.20 for the case of $i \geq 1$.

Lemma 4.4.33. *Suppose that Lemma 4.4.20 holds at the end of round $i-1$ for $i \geq 1$. Then it holds still at the end of round i.*

Proof. In round i, it is easy to verify that the Lemma 4.4.20 is true in the beginning of the sub-procedure *Conflict set construction.* Now let the original instance apply all the fake orientations and let the new instance mimic, using Lemma 4.4.30. Next let the new instance finish off its fake orientations arbitrarily. It is easy to see that Lemma 4.4.20 holds at the end of round i.

□

We now handle the more difficult case of round 0. Unlike the later rounds, Lemma 4.4.20 does not hold in the beginning of the sub-procedure *Conflict set construction*: the set of overloaded nodes can be different in the two instances and the conflict set in the new instance may not be a superset of the conflict set in the original instance.

In the following, we postpone the fake orientations of the original instance and just let the new instance perform some fake orientations until Lemma 4.4.20(2)(3) hold.

Lemma 4.4.34. *Consider the beginning of the sub-procedure* Conflict set construction *in round 0. In the new instance, as long as an edge $e = vu \in E(\mathbb{C}'_0)$ remains neutral and v is part of $\mathbb{C}^{\dagger}_0$, direct e toward u. Then finally, $\mathbb{C}'_0 \subseteq C^{\dagger}_0$.*

Proof. [7] Consider a connected component H in the induced subgraph $G_{\mathbb{R}}[\mathbb{C}'_0]$, and let us first assume H is a tree. It is easy to see that because in the original instance every node $v \in H$ can follow a directed path to some overloaded node in H, v cannot receive all incident edges in H without becoming overloaded. Suppose the lemma does not hold and consider a maximal tree $\overline{T} \subseteq H$ remaining outside of $\mathbb{C}^{\dagger}_0$. In the new instance, all edges of H connecting $\overline{T}$ to the rest of the nodes in $H \backslash \overline{T}$ are directed toward $\overline{T}$. By induction, we can show that there is a node $v \in \overline{T}$ which receives all its incident edges in H, implying that $v \in \mathbb{C}^{\dagger}_0$, a contradiction.

[7] The proof here is very similar to the proof of Lemma 4.4.22, Case 1. So we only sketch the ideas.

If H is a cycle, observe that at least one node in H must be overloaded in the new instance and hence part of $\mathbb{C}_0^\dagger$. Now we can proceed as before.

□

Lemma 4.4.35. *In round 0, suppose that both instances are in the sub-procedure* Conflict set construction *and* $\mathbb{C}'_0 \subseteq \mathbb{C}_0^\dagger$. *In the new instance, as long as there is an edge* $e = vu \notin E(\mathbb{C}'_0)$ *so that (1) it is directed toward* u *in the original instance, (2) it is currently neutral in the new instance, and (3)* $v \in \mathbb{C}_0^\dagger$ *and* $u \notin \mathbb{C}_0^\dagger$, *let* e *be directed toward* u *in the new instance. Then finally, an edge* $e \notin E(\mathbb{C}_0^\dagger)$, *if directed in the original instance, is directed the same way in the new instance.*

Proof. Let $E_{\mathfrak{i}}(v)$ and $E_{\mathfrak{o}}(v)$ denote the current set of incoming and outgoing edges of a node $v \notin \mathbb{C}'_0$ in the original instance. In the new instance, after the fake orientations required in the lemma, if every edge in $E_{\mathfrak{i}}(v)$ is directed toward v, then every edge in $E_{\mathfrak{o}}(v)$ must be directed away from v, otherwise v is overloaded.

We now prove the lemma by contradiction. Suppose that edge $e_0 = v_0 u \notin E(\mathbb{C}_0^\dagger)$ is directed toward u in the original instance while it is neutral or directed toward v_0 in the new instance after the fake orientations required in the lemma. In both cases $v_0 \notin \mathbb{C}_0^\dagger$ (hence $v_0 \notin \mathbb{C}'_0$) and v_0 is not overloaded. So there is an edge $e_1 = v_1 v_0 \in E_{\mathfrak{i}}(v_0)$ that is neutral or directed away from v_0 in the new instance after the fake orientations. As before, $v_1 \notin \mathbb{C}_0^\dagger$. Repeating this argument, we conclude that the entire system is a cycle with no node in $\mathbb{C}_0^\dagger$ (hence also not in $\mathbb{C}'_0$), whose edges are all directed, say clockwise, in the original instance. Furthermore, in the new instance, each edge in the cycle is either neutral or directed counter-clockwise. Clearly, for at least one edge xy in the cycle, it holds that $dl(x) + pl(x) + w_{xy} > (5/3 + \beta/3)t$. Since x is not overloaded, this edge must have the same orientation (namely toward y) in both instances, a contradiction. □

By Lemmas 4.4.34 and 4.4.35, item (2) and (3) of Lemma 4.4.20 hold, and we can apply Lemma 4.4.30 to finish off all the remaining fake orientations in both instances while maintaining Lemma 4.4.20(2)(3).

The last thing to prove is that $\mathbb{A}'_0 \subseteq \mathbb{A}_0^\dagger$ after the activation rules are applied to both instances. If a node v is overloaded in the original instance, by Lemma 4.4.1, either its own pebble and dedicated load is already more than $(5/3+\beta/3)t$, or it has a child $u \in \mathbb{C}'_0$ so that $pl(v) + dl(v) + w_{vu} > (5/3+\beta/3)t$. Thus, in the new instance, v is either overloaded, or (as $u \in \mathbb{C}'_0 \subseteq \mathbb{C}_0^\dagger$) becomes a child of u and is activated by Rule 1. Furthermore, if a node v is activated by Rule 1 in the original instance, then it has a father $u \in \mathbb{C}'_0$ satisfying $dl(v) + pl(v) + w_{vu} > (5/3 + \beta/3)t$. As u is also part of $\mathbb{C}_0^\dagger$ in the new instance, either v is overloaded, or it is again activated by Rule 1. So we are sure Type A and Type B nodes of the original instance in $\mathbb{A}'_0$ must be part of $\mathbb{A}_0^\dagger$. Finally, as Lemma 4.4.20(2) holds, nodes of $\mathbb{A}'_0$ activated by Rule 2 must also be part of $\mathbb{A}_0^\dagger$. This completes the proof of round 0 and the entire proof of Lemma 4.4.20.

Bibliography

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

[2] S. Albers and A. Antoniadis. Race to idle: New algorithms for speed scaling with a sleep state. *ACM Trans. Algorithms*, 10(2):9:1–9:31, 2014.

[3] S. Albers and H. Fujiwara. Energy-efficient algorithms for flow time minimization. In *23rd Annual Symposium on Theoretical Aspects of Computer Science (STACS '06)*, pages 621–633. Springer, 2006.

[4] S. Albers and H. Fujiwara. Energy-efficient algorithms for flow time minimization. *ACM Trans. Algorithms*, 3(4), 2007.

[5] S. Albers, F. Müller, and S. Schmelzer. Speed scaling on parallel processors. In *Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '07)*, pages 289–298. ACM, 2007.

[6] S. Albers, A. Antoniadis, and G. Greiner. On multi-processor speed scaling with migration: extended abstract. In *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '11)*, pages 279–288. ACM, 2011.

[7] E. Angel, E. Bampis, and V. Chau. Throughput maximization in the speed-scaling setting. In *31st International Symposium on Theoretical Aspects of Computer Science (STACS '14)*, pages 53–62. Schloss Dagstuhl - LZI, 2014.

[8] A. Antoniadis and C.-C. Huang. Non-preemptive speed scaling. In *Algorithm Theory - 13th Scandinavian Symposium and Workshops (SWAT '12)*, pages 249–260. Springer, 2012.

[9] A. Antoniadis, N. Barcelo, M. E. Consuegra, P. Kling, M. Nugent, K. Pruhs, and M. Scquizzato. Efficient computation of optimal energy and fractional weighted flow trade-off schedules. In *31st International Symposium on Theoretical Aspects of Computer Science (STACS '14)*, pages 63–74. Schloss Dagstuhl - LZI, 2014.

[10] A. Antoniadis, C.-C. Huang, and S. Ott. A fully polynomial-time approximation scheme for speed scaling with sleep state. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '15)*, pages 1102–1113. SIAM, 2015.

[11] Y. Asahiro, J. Jansson, E. Miyano, H. Ono, and K. Zenmyo. Approximation algorithms for the graph orientation minimizing the maximum weighted outdegree. *J. Comb. Optim.*, 22(1):78–96, 2011.

[12] Y. Azar, L. Epstein, Y. Richter, and G. J. Woeginger. All-norm approximation algorithms. *J. Algorithms*, 52(2):120–133, 2004.

[13] P. Bailis, V. J. Reddi, S. Gandhi, D. Brooks, and M. Seltzer. Dimetrodon: Processor-level preventive thermal management via idle cycle injection. In *Proceedings of the 48th Design Automation Conference (DAC '11)*, pages 89–94. ACM, 2011.

[14] E. Bampis, C. Dürr, F. Kacem, and I. Milis. Speed scaling with power down scheduling for agreeable deadlines. *Sustainable Computing: Informatics and Systems*, 2(4):184–189, 2012.

[15] E. Bampis, A. Kononov, D. Letsios, G. Lucarelli, and I. Nemparis. From preemptive to non-preemptive speed-scaling scheduling. In *Computing and Combinatorics, 19th International Conference (COCOON '13)*, pages 134–146. Springer, 2013.

[16] E. Bampis, A. Kononov, D. Letsios, G. Lucarelli, and M. Sviridenko. Energy efficient scheduling and routing via randomized rounding. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS '13)*, pages 449–460. Schloss Dagstuhl - LZI, 2013.

[17] E. Bampis, D. Letsios, and G. Lucarelli. Speed-scaling with no preemptions. In *Algorithms and Computation - 25th International Symposium (ISAAC '14)*, pages 259–269. Springer, 2014.

[18] N. Bansal, T. Kimbrel, and K. Pruhs. Speed scaling to manage energy and temperature. *J. ACM*, 54(1):3:1–3:39, 2007.

[19] N. Bansal, K. Pruhs, and C. Stein. Speed scaling for weighted flow time. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '07)*, pages 805–813. SIAM, 2007.

[20] N. Bansal, H.-L. Chan, T. W. Lam, and L.-K. Lee. Scheduling for speed bounded processors. In *Automata, Languages and Programming, 35th International Colloquium (ICALP '08)*, pages 409–420. Springer, 2008.

[21] N. Bansal, H.-L. Chan, and K. Pruhs. Speed scaling with an arbitrary power function. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '09)*, pages 693–701. SIAM, 2009.

[22] N. Bansal, H.-L. Chan, and K. Pruhs. Speed scaling with a solar cell. *Theor. Comput. Sci.*, 410(45):4580–4587, 2009.

[23] N. Bansal, H.-L. Chan, D. Katz, and K. Pruhs. Improved bounds for speed scaling in devices obeying the cube-root rule. *Theory of Computing*, 8(1):209–229, 2012.

[24] N. Bansal, H.-L. Chan, and K. Pruhs. Speed scaling with an arbitrary power function. *ACM Trans. Algorithms*, 9(2):18:1–18:14, 2013.

[25] N. Bansal, H.-L. Chan, and K. Pruhs. Speed scaling with an arbitrary power function. *ACM Trans. Algorithms*, 9(2):18, 2013.

[26] P. Baptiste. Scheduling unit tasks to minimize the number of idle periods: A polynomial time algorithm for offline dynamic power management. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '06)*, pages 364–367. SIAM, 2006.

[27] P. Baptiste, M. Chrobak, and C. Dürr. Polynomial-time algorithms for minimum energy scheduling. *ACM Trans. Algorithms*, 8(3):26:1–26:29, 2012.

[28] B. D. Bingham and M. R. Greenstreet. Energy optimal scheduling on multiprocessors with migration. In *IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA '08)*, pages 153–161. IEEE Computer Society, 2008.

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

[30] D. Brooks, P. Bose, S. Schuster, H. M. Jacobson, P. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. V. Zyuban, M. Gupta, and P. W. Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6):26–44, 2000.

[31] D. Chakrabarty, S. Khanna, and S. Li. On $(1, \epsilon)$-restricted assignment makespan minimization. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '15)*, pages 1087–1101. SIAM, 2015.

[32] H.-L. Chan, J. W.-T. Chan, T.-W. Lam, L.-K. Lee, K.-S. Mak, and P. W. H. Wong. Optimizing throughput and energy in online deadline scheduling. *ACM Trans. Algorithms*, 6(1):10:1–10:22, Dec. 2009.

[33] J.-J. Chen, T.-W. Kuo, and H.-I. Lu. Power-saving scheduling for weakly dynamic voltage scaling devices. In *Algorithms and Data Structures, 9th International Workshop (WADS '05)*, pages 338–349. Springer, 2005.

[34] V. Cohen-Addad, Z. Li, C. Mathieu, and I. Milis. Energy-efficient algorithms for non-preemptive speed-scaling. In *Approximation and Online Algorithms - 12th International Workshop (WAOA '14)*, pages 107–118. Springer, 2014.

[35] E. D. Demaine, M. Ghodsi, M. Hajiaghayi, A. S. Sayedi-Roshkhar, and M. Zadimoghaddam. Scheduling to minimize gaps and power consumption. *J. Scheduling*, 16(2):151–160, 2013.

[36] T. Ebenlendr, M. Krčál, and J. Sgall. Graph balancing: A special case of scheduling unrelated parallel machines. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '08)*, pages 483–490. SIAM, 2008.

[37] T. Ebenlendr, M. Krčál, and J. Sgall. Graph balancing: A special case of scheduling unrelated parallel machines. *Algorithmica*, 68:62–80, 2014.

[38] K. Fang, N. A. Uhan, F. Zhao, and J. W. Sutherland. Scheduling on a single machine under time-of-use electricity tariffs. *Annals of Operations Research*, pages 1–29, 2015.

[39] M. Gairing, T. Lücking, M. Mavronicolas, and B. Monien. Computing nash equilibria for scheduling on restricted parallel links. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC '04)*, pages 613–622. ACM, 2004.

[40] M. Gairing, B. Monien, and A. Wocalw. A faster combinatorial approximation algorithm for scheduling unrelated parallel machines. *Theor. Comput. Sci.*, 380 (1-2):87–99, 2007.

[41] A. Gandhi, M. Harchol-Balter, R. Das, and C. Lefurgy. Optimal power allocation in server farms. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '09)*, pages 157–168. ACM, 2009.

[42] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[43] M. Garrett. Powering down. *Queue*, 5(7):16–21, 2007.

[44] Y. A. Gonczarowski and M. Tennenholtz. Noncooperative market allocation and the formation of downtown. *CoRR*, abs/1403.7536, 2014.

[45] X. Han, T. W. Lam, L.-K. Lee, I. K.-K. To, and P. W. H. Wong. Deadline scheduling and power management for speed bounded processors. *Theor. Comput. Sci.*, 411(40-42):3587–3600, 2010.

[46] C.-C. Huang and S. Ott. New results for non-preemptive speed scaling. In *Mathematical Foundations of Computer Science 2014 - 39th International Symposium (MFCS '14)*, pages 360–371. Springer, 2014.

[47] C.-C. Huang and S. Ott. A combinatorial approximation algorithm for graph balancing with light hyper edges. *CoRR*, abs/1507.07396, 2015.

[48] S. Irani and K. Pruhs. Algorithmic problems in power management. *SIGACT News*, 36(2):63–76, 2005.

[49] S. Irani, S. K. Shukla, and R. K. Gupta. Algorithms for power savings. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '03)*, pages 37–46. SIAM, 2003.

[50] S. Irani, S. Shukla, and R. Gupta. Algorithms for power savings. *ACM Trans. Algorithms*, 3(4), 2007.

[51] ITWatchDogs. Are heat & humidity hurting your equipment? *Processor*, 36 (20), 2014.

[52] K. Jain and V. V. Vazirani. Approximation algorithms for metric facility location and k-median problems using the primal-dual schema and lagrangian relaxation. *J. ACM*, 48(2):274–296, 2001.

[53] S. M. Johnson. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, 1(1):61–68, 1954.

[54] S. G. Kolliopoulos and Y. Moysoglou. The 2-valued case of makespan minimization with assignment constraints. *Information Processing Letters*, 113 (1-2):39–43, 2013.

[55] P. Kosmol and D. Müller-Wichards. *Optimization in Function Spaces: With Stability Considerations in Orlicz Spaces*. De Gruyter, 2011.

[56] I. Koutsopoulos and L. Tassiulas. Control and optimization meet the smart power grid: scheduling of power demands for optimal energy management. In *2nd International Conference on Energy-Efficient Computing and Networking (e-Energy '11)*, pages 41–50. ACM, 2011.

[57] G. Kumar and S. Shannigrahi. NP-hardness of speed scaling with a sleep state. *CoRR*, abs/1304.7373, 2013.

[58] J. Lenstra, D. Shmoys, and E. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46:256–271, 1990.

[59] J. Leung and C. Li. Scheduling with processing set restrictions: A survey. *International Journal of Production Economics*, 116:251–262, 2008.

[60] M. Li. Approximation algorithms for variable voltage processors: Min energy, max throughput and online heuristics. *Theor. Comput. Sci.*, 412(32):4074–4080, 2011.

[61] M. Li and F. F. Yao. An efficient algorithm for computing optimal discrete voltage schedules. In *Mathematical Foundations of Computer Science 2005, 30th International Symposium (MFCS '05)*, pages 652–663. Springer, 2005.

[62] M. Li, B. J. Liu, and F. F. Yao. Min-energy voltage allocation for tree-structured tasks. In *Computing and Combinatorics, 11th Annual International Conference (COCOON '05)*, pages 283–296. Springer, 2005.

[63] M. Li, A. C. Yao, and F. F. Yao. Discrete and continuous min-energy schedules for variable voltage processors. *PNAS*, 103(11):3983–3987, 2006.

[64] M. Li, F. F. Yao, and H. Yuan. An $O(n^2)$ algorithm for computing optimal continuous voltage schedules. *CoRR*, abs/1408.5995, 2014.

[65] D. G. Luenberger. *Optimization by Vector Space Methods*. John Wiley & Sons, Inc., 1997.

[66] G. Muratore, U. M. Schwarz, and G. J. Woeginger. Parallel machine scheduling with nested job assignment restrictions. *Oper. Res. Lett.*, 38(1):47–50, 2010.

[67] M. Negrean and R. Ernst. Response-time analysis for non-preemptive scheduling in multi-core systems with shared resources. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES '12)*, pages 191–200. IEEE Computer Society, 2012.

[68] C. N. Potts and V. A. Strusevich. Fifty years of scheduling: a survey of milestones. *JORS*, 60(S1), 2009.

[69] A. Raghavan, L. Emurian, L. Shao, M. C. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. K. Martin. Utilizing dark silicon to save energy with computational sprinting. *IEEE Micro*, 33(5):20–28, 2013.

[70] J. Sgall. Private communication, 2015.

[71] E. V. Shchepin and N. Vakhania. An optimal rounding gives a better approximation for scheduling unrelated machines. *Oper. Res. Lett.*, 33(2):127–133, 2005.

[72] D. R. Smith. *Variational Methods in Optimization*. Dover Books on Mathematics. Dover Publications, 1998.

[73] O. Svensson. Santa claus schedules jobs on unrelated machines. *SIAM J. Comput.*, 41(5):1318–1341, 2012.

[74] N. K. Thang. Lagrangian duality in online scheduling with resource augmentation and speed scaling. In *Algorithms - 21st Annual European Symposium (ESA '13)*, pages 755–766. Springer, 2013.

[75] V. V. Vazirani. *Approximation Algorithms*. Springer, 2001.

[76] J. Verschae and A. Wiese. On the configuration-lp for scheduling on unrelated machines. *J. Scheduling*, 17(4):371–383, 2014.

[77] A. Wierman, L. L. H. Andrew, and A. Tang. Power-aware speed scaling in processor sharing systems: Optimality and robustness. *Perform. Eval.*, 69(12): 601–622, 2012.

[78] D. P. Willamson and D. Shmoys. *The design of approximation algorithms.* Cambridge University Press, 2010.

[79] F. F. Yao, A. J. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *36th Annual Symposium on Foundations of Computer Science (FOCS '95)*, pages 374–382. IEEE Computer Society, 1995.